ISBN 978-1-939133-28-1

Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22) *Carlsbad, CA, USA July 11–13, 2022*

conference

proceedings

16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)

Carlsbad, CA, USA
July 11–13, 2022

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with ACM SIGOPS

OSDI '22 Sponsors

Platinum Sponsor

amazon

Gold Sponsor

Google

Silver Sponsors

Akamai

Apple

FUTUREWEI Technologies

Meta

Microsoft

MOLOCO

Bronze Sponsors

Baidu 百度

exotanium

IBM

TWO SIGMA

vmware

Open Access Sponsor

NetApp

USENIX Supporters

USENIX Patrons

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

USENIX Benefactors

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

USENIX Partners

Blameless • Lightstep • Top10VPN

Open Access Supporter

Google

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation

July 11–13, 2022
Carlsbad, CA, USA

ISBN 978-1-939133-28-1

Conference Organizers

Program Co-Chairs

Marcos K. Aguilera, *VMware Research*
Hakim Weatherspoon, *Cornell University and Exotanium, Inc.*

Program Committee

Atul Adya, *Databricks*
Nitin Agrawal, *ThoughtSpot*
Marcos K. Aguilera, *VMware Research*
Deniz Altinbüken, *Google*
Behnaz Arzani, *Microsoft Research*
Mahesh Balakrishnan, *Facebook*
Oana Balmau, *McGill University*
Andrew Baumann, *Microsoft Research*
Adam Belay, *MIT CSAIL*
Ranjita Bhagwan, *Microsoft Research*
Annette Bieniusa, *Technische Universität Kaiserslautern*
Ken Birman, *Cornell University*
Rodrigo Bruno, *INESC-ID and Instituto Superior Técnico, University of Lisbon*
Irina Calciu, *Graft*
George Candea, *EPFL*
Marco Canini, *KAUST*
Miguel Castro, *Microsoft*
Rong Chen, *Shanghai Jiao Tong University*
Byung-Gon Chun, *Seoul National University and FriendliAI*
Asaf Cidon, *Columbia University*
Landon Cox, *Microsoft Research*
Natacha Crooks, *University of California, Berkeley*
Murat Demirbas, *Amazon Web Services*
Yufei Ding, *University of California, Santa Barbara*
Roxana Geambasu, *Columbia University*
Ashvin Goel, *University of Toronto*
Haryadi S. Gunawi, *University of Chicago*
Indranil Gupta, *University of Illinois at Urbana–Champaign*
Andreas Haeberlen, *University of Pennsylvania*
Wenjun Hu, *Yale University*
Rüdiger Kapitza, *Technische Universität Braunschweig*
Manos Kapritsos, *University of Michigan*
Baris Kasikci, *University of Michigan*
Ana Klimovic, *ETH Zurich*
Eddie Kohler, *Harvard University*
Kevin Kornegay, *Morgan State University*
Dejan Kostić, *KTH Royal Institute of Technology*
Philip Levis, *Stanford University*
Jialin Li, *National University of Singapore*
Jonathan Mace, *Max Planck Institute for Software Systems (MPI-SWS)*
Ratul Mahajan, *University of Washington and Intentionet*
Petros Maniatis, *Google Research*
Z. Morley Mao, *University of Michigan*
Kathryn S McKinley, *Google*
Neha Narula, *Massachusetts Institute of Technology*
Ravi Netravali, *Princeton University*
Jason Nieh, *Columbia University*
Cristina Nita-Rotaru, *Northeastern University*
Shadi Noghabi, *Microsoft Research*
Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*
Aurojit Panda, *NYU*
Amar Phanishayee, *Microsoft Research*
Peter Pietzuch, *Imperial College London*
Luis Rodrigues, *INESC-ID and Instituto Superior Técnico, University of Lisbon*
Christopher Rossbach, *The University of Texas at Austin and Katana Graph*
Malte Schwarzkopf, *Brown University*
Marco Serafini, *University of Massachusetts Amherst*
Marc Shapiro, *Inria and UPMC-LIP6*
Ji-Yong Shin, *Northeastern University*
Liuba Shrira, *Brandeis University*
Vishal Shrivastav, *Purdue University*
Alex C. Snoeren, *University of California, San Diego*
Robert Soulé, *Yale University*
Phillip Stanley-Marbell, *University of Cambridge*
Swami Sundararaman, *Pyxeda AI*
Adriana Szekeres, *VMware Research*
Amy Tai, *VMware Research*
Doug Terry, *Amazon Web Services*
Dan Tsafrir, *Technion—Israel Institute of Technology and VMware Research*
Ymir Vigfusson, *Emory University*
Rashmi Vinayak, *Carnegie Mellon University*
Hakim Weatherspoon, *Cornell University and Exotanium, Inc.*
Bernard Wong, *University of Waterloo*
Tim Wood, *George Washington University*
Gala Yadgar, *Technion—Israel Institute of Technology*
Ding Yuan, *University of Toronto*
Gerd Zellweger, *VMware Research*
Yiying Zhang, *University of California, San Diego*

Poster Session Co-Chairs

Natacha Crooks, *University of California, Berkeley*
Adriana Szekeres, *VMware Research*

Steering Committee

Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Angela Demke Brown, *University of Toronto*
Jason Flinn, *Facebook*
Casey Henderson, *USENIX Association*
Jon Howell, *VMware Research*
Kimberly Keeton
Hank Levy, *University of Washington*
Jay Lorch, *Microsoft Research*
Shan Lu, *University of Chicago*
James Mickens, *Harvard University*
Timothy Roscoe, *ETH Zurich*
Margo Seltzer, *University of British Columbia*
Geoff Voelker, *University of California, San Diego*

External Reviewers

Anirudh Badam
Can Cebeci
Ranveer Chandra
Adrian Chiu
Mosharaf Chowdhury
Rishikesh Devsot
Ittay Eyal
Chris Hawblitzel
Jon Howell
Kevin Hsieh
David Irwin
Michael Isard
Rishabh Iyer
Ruibin Li
Yatin A. Manerkar
Beomseok Nam
Kexin Pei
Solal Pirelli
Laurent Prosperi
Shaz Qadeer
Xiang Ren
Adrian Sampson
Foteini Strati
Caroline Trippel
Rui Wang
Haiqi Xu
Lei Yan

Message from the OSDI '22 Program Co-Chairs

Dear colleagues,

Welcome to the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22).

This year OSDI '22 is co-located with the 2022 USENIX Annual Technical Conference (ATC '22). We are holding the joint conference in a hybrid format, with the option of virtual or physical participation. We are excited to return to a physical event, in Carlsbad, California, after a two-year hiatus. Due to Covid, the previous two instances of OSDI (2020 and 2021) were held only virtually, and we have made the best of it. That has made OSDI more accessible but less interactive. This year, with the hybrid format, we hope to get the best of both worlds. We have encouraged presenters to attend in person if they can, while attendants have the option of joining in person or virtually.

OSDI '22 received 253 submissions and accepted 49 of them for a 19.4% acceptance rate. As in the last OSDI, we did not allow the program chairs to submit papers. Due to the historically high number of submissions, we recruited a large PC of 76 members in addition to the two chairs. PC members included academics, industrial researchers, and industrial practitioners. We also recruited a few additional people to serve as an external review committee to provide additional coverage of expertise if necessary. We are grateful to all the committee members for their hard work that was essential to the success of OSDI '22.

The program committee reviewed the submissions in two rounds. In the first round, papers received three reviews. About 29% of papers were then rejected based on these reviews, while the others advanced to the second round. In the second round, papers received at least two additional reviews and, in some cases, we solicited additional input from expert external reviewers. We discussed these papers online and reached a decision for another 52% of the submitted papers. The remaining papers were discussed and decided over a three-day online PC meeting. Each of the accepted papers was shepherded by a PC member to help the authors address the reviewers' comments in the camera-ready version. In total, we produced over 1130 reviews and 2900 online comments, representing an enormous amount of work. We estimate the human cost of evaluating the papers to be above $1M based on an average of 4.5 hours per review (to read the paper, write the report, and discuss it in the group) and a cost of $200/hour per reviewer. This is a significant price to the community and as such we must optimize it going forward while maintaining the high quality that OSDI is known for. Toward that goal, OSDI '22 experimented with some approaches that make better use of the collective effort.

In particular, OSDI '22 introduced a change to the reviewing process: the option to revise and resubmit. This option was given to a small number of papers that were rejected but that reviewers felt would have been accepted if authors could address a list of objective issues. This list was provided to authors so they can work on a revision of the paper. The revision will then be re-evaluated by the same reviewers if possible, for publication in OSDI next year, OSDI '23, based on how well the authors address the issues on the list. This year, only six papers were given the option to revise-and-resubmit but this number should increase in future years if OSDI continues with this practice. All six papers have decided to resubmit. These revise-and-resubmit papers are now under evaluation.

After finalizing the program, we proceeded to decide the Jay Lepreau Best Paper Awards. We asked all PC members to nominate papers. We next created a short list based on the nominations, the reviews, and the paper themselves. We then selected a small set of PC members that were not conflicting with any of the papers in the short list, and we asked them to score each paper. Based on the nominations, reviews, and scores, the best papers were selected.

OSDI '22 had an artifact-evaluation process organized by three co-chairs: Anuj Kalia, Neeraja J. Yadwadkar, and Chengyu Zhang. Of the 49 papers accepted to OSDI '22, 35 had artifacts submitted by their authors. Of those 33 earned the "Available" badge, 31 artifacts earned the "Functional" badge and 27 earned the most challenging "Results Reproduced" badge. For more details, see the Message from the OSDI '22 Artifact Evaluation Committee Co-Chairs.

OSDI '22 had a poster submission process organized by Natacha Crooks and Adriana Szekeres. Submissions were open to all, and authors of papers accepted to OSDI '22 were encouraged to submit a poster. We accepted 52 posters. For more details, see the Message from the OSDI '22 Poster Co-Chairs.

As PC co-chairs, we rely on many people to make OSDI '22 a success, to whom we are grateful. We thank the authors for choosing to submit their work to OSDI. We thank the program committee and external reviewers for their arduous work in reviewing and discussing the submissions. We thank the co-chairs and all members of the Artifact Evaluation Committee, who conducted thorough evaluations. We also thank the co-chairs of the poster committee, who identified high-quality posters for the conference. We thank Jiri Schindler and Noa Zilberman, the program co-chairs of ATC '22, for coordinating with us efficiently, productively, and enjoyably. We thank the USENIX staff, who have been fundamental in organizing OSDI '22 as we transition to a hybrid format. The logistics of the online PC meeting were facilitated by PhD student Daniel Amir, whose assistance we greatly appreciate. Finally, OSDI wouldn't be what it is without our attendees. Thank you for listening to our speakers, asking challenging and insightful questions, and sharing your ideas with others.

We hope you will find OSDI '22 interesting, educational, and inspiring!

Marcos K. Aguilera, *VMware*
Hakim Weatherspoon, *Cornell University and Exotanium, Inc.*
OSDI '22 Program Co-Chairs

Message from the OSDI '22 Artifact Evaluation Committee Co-Chairs

We are happy to report about the OSDI '22 artifact evaluation process. This is the third time that OSDI conducted such a process and we hope to keep improving it so that artifact evaluation will become more common in our community's conferences. This year, the OSDI '22 artifact evaluation process is combined with USENIX ATC '22. The combined artifact evaluation committee consists of 118 artifact reviewers from academia and industry.

Process
We continued to use the three-badge approach (vs. the single-badge approach) from OSDI '21 evaluation and these three badges include:

- **Artifacts Available:** To earn this badge, the AEC must judge that the artifacts associated with the paper have been made available for retrieval, permanently and publicly.
- **Artifacts Functional:** To earn this badge, the AEC must judge that the artifacts conform to the expectations set by the paper in terms of functionality, usability, and relevance.
- **Results Reproduced:** To earn this badge, the AEC must judge that they can use the submitted artifacts to obtain the main results presented in the paper.

Evaluation
In the evaluation process, each artifact was evaluated by 3 reviewers. The evaluation process had two key phases: the kick-the-tires phase and the in-depth evaluation phase. During the kick-the-tires phase, reviewers made a quick first pass over all assignments to identify and report obvious problems and communicated them with the authors. After the kick-the-tires phase, reviewers evaluated each assignment thoroughly and wrote detailed reviews. Finally, reviewers coordinated and communicated with fellow AEC members and decided which badges should be awarded to each artifact.

Results
OSDI '22 accepted 49 papers and 35 papers participated in the AE. Of the 35 submitted artifacts:

- 33 artifacts received the Artifacts Available badge (94%).
- 31 artifacts received the Artifacts Functional badge (88%).
- 27 artifacts received the Results Reproduced badge (77%).

Key Takeaways
Our experience shows that after the kick-the-tires response period, reviewers can still encounter technical problems that obstruct their evaluation. We suggest future AEC chairs extend the kick-the-tires response period and encourage more interaction between reviewers and authors during the period.

Finally, we deeply thank the authors and the AEC committee for all their efforts in making the OSDI '22 artifact evaluation possible.

Anuj Kalia, *Microsoft*
Neeraja J. Yadwadkar, *University of Texas at Austin*
Chengyu Zhang, *ETH Zurich*
OSDI '22 Artifact Evaluation Committee Co-chairs

Message from the OSDI '22 Poster-Session Co-Chairs

We are happy to report about the OSDI '22 poster session. We accepted a total of 52 posters. 28 of these posters correspond to accepted OSDI papers. The remaining 24 were independent submissions. We reviewed these posters for conference fit and clarity of problem exposition and motivation. We intentionally encouraged posters describing early work, as well as more mature projects.

We look forward to lively and interesting discussions at the OSDI '22 poster session.

Natacha Crooks, UC Berkeley
Adriana Szekeres, VMware Research
OSDI'22 Poster Co-Chairs

16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)

July 11–13, 2022

Carlsbad, CA, USA

Monday, July 11

Distributed Storage and Far Memory

Bugs

Persistent Memory

Formal Verification

Machine Learning 2

Wednesday, July 13

Isolation and OS Services

Security and Private Messaging

Managed Languages

Recommenders and Pattern Mining

Owl: Scale and Flexibility in Distribution of Hot Content

Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko,
Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, Fang Zhou
Meta

Abstract

Owl provides high-fanout distribution of large data objects to hosts in Meta's private cloud. Owl combines a decentralized data plane based on ephemeral peer-to-peer distribution trees with a centralized control plane in which tracker services maintain detailed metadata about peers, their cache state, and ongoing downloads. In Owl, peer nodes are simple state machines and centralized trackers decide from where each peer should fetch data, how they should retry on failure, and which data they should cache and evict. Owl trackers provide a highly-flexible and configurable policy interface that customizes and optimizes behavior for widely-varying distribution use cases. In contrast to prior assumptions about peer-to-peer distribution, Owl shows that centralizing the control plan is not a barrier to scalability: Owl distributes over 800 petabytes of data per day to millions of client processes. Owl improves download speeds by a factor of 2–3 over both BitTorrent and a prior decentralized static distribution tree used at Meta, while supporting 106 use cases that collectively employ 55 different distribution policies.

1 Introduction

Within Meta's private cloud, efficient distribution of large, hot content to end hosts is an increasingly important requirement. Three dimensions express the scope of the task: (1) scale: the same content may be read by anywhere from a handful of clients to millions of processes running in data centers around the globe, (2) size: objects to be distributed range from 1 MB to a few terabytes, and (3) hotness: all clients may read an object within a few seconds of each other, or their reads may be spread over hours. At Meta, executables, code artifacts, AI models, and search indexes are content types commonly distributed within this scope.

Distribution requirements are exacting. First, content distribution must be *fast*: the predictive value of AI models decreases over time, and slow executable delivery increases downtime and delays deploying fixes. We expect to provide data at a rate bounded by either the available network bandwidth of the reading host or by the available write bandwidth of its storage media.

Second, content distribution must be *efficient*. One dimension of efficiency is scalability, i.e., the number of clients that can have their distribution needs met by a given number of servers. Another dimension is network usage, which we measure both in terms of bytes transmitted and communication locality (e.g., an in-rack data transfer is less costly than a cross-region transfer) . A final dimension of efficiency is resource usage on client machines; e.g., CPU cycles, memory, and disk I/O. Not only should we use as few resources as possible, but we should also adjust for their relative importance on different clients; e.g., some services are memory-constrained, while others are CPU-constrained or cannot afford to write to disk.

Finally, content distribution must be *reliable*. Reliability is measured as the percentage of download requests that the distribution system satisfies within a latency SLA. Operational ease-of-management is an oft-overlooked prerequisite for high reliability. In a production environment, workloads change, dependent service and infrastructure may have partial outages, and performance faults in which a dependency doesn't meet its own SLA are not uncommon. In order to maintain a high SLA for distribution, engineers need to be alerted quickly about such events, and they need a clear picture of operational health for each client type. Finally, they need simple knobs that adjust behavior when reliability, speed, or efficiency starts to degrade in order to restore operational health quickly.

Prior to our work, Meta used at least three different systems for large content distribution. No prior solution met all of the above requirements. We identified two root causes: (1) no prior system struck the correct balance between decentralization and centralization, and (2) no prior system was sufficiently flexible to meet all of the requirements of the many different types of services at Meta that require content distribution.

Meta previously implemented highly-centralized distribution via hierarchical caching, in which clients download content from first-level caches on remote hosts. These caches, in turn, handle cache misses by reading from other caches, with the final layer of the hierarchy being a distributed storage system. Hierarchical caching is inefficient for hot content distribution, and it is difficult to scale. Meta needed dedicated hosts in great quantity to implement the cache hierarchy. The number of hosts increased to keep pace with growth in workloads from services consuming the data and with growth in the number of reading clients. Load spikes caused by hot con-

tent were a continual problem: strict quotas were necessary to protect the centralized caches. However, readers of hot content were frequently throttled because they exceeded their quotas. In general, provisioning for transient spikes caused by hot content and setting quotas appropriately was quite challenging.

Meta also used two highly-decentralized systems: a location-aware BitTorrent [7] implementation and a static peer-to-peer distribution tree based on consistent hashing, which we will refer to as StaticTree. In both cases, a peer is any process that wishes to download data, and there are millions of such processes at Meta. The decentralized systems scaled much better than hierarchical caching, but they brought their own problems. First, because each peer made distribution decisions based on local information, resource efficiency and speed could be poor; e.g., with each peer making independent caching decisions, the collection of peers could retain either more or less copies of a data object than necessary. Perhaps more importantly, these decentralized solutions were difficult to operate. Engineers could not get a clear picture of health and status without aggregating data from large numbers of peers. Each peer had a different and limited view of the state of distribution, so it was often hard to tell whether or not a collection of peers was making good decisions. In general, it was very hard to reason about system-wide correctness or efficiency.

In summary, decentralized systems were inefficient and difficult to operate, while centralized systems scaled poorly. As a result, we chose to create a new, split design with a decentralized data plane and a centralized control plane. The decentralized data plane streams data from sources to clients via a distribution tree. However, its trees are *ephemeral*, i.e., each tree tracks a single data chunk, and each edge in a tree persists only while the chunk is being transferred from a source to a peer.

The design realizes a mechanism-policy split. Peers are simple and provide the mechanism for caching and transferring data chunks. The centralized control plane makes all detailed policy decisions about distribution, e.g., from where peers should get each chunk of content, when and how they should cache content, and how they should retry failed downloads. The control plane is implemented by a small set of *trackers* [1]. Trackers have a complete picture of the distribution state; e.g., which data each peer is downloading, where these peers are located, and which chunks are in each peer's cache. Detailed state enables trackers to make highly-optimized decisions about data placement and distribution that minimize the use of expensive network links and maximize cache hit rate. Centralizing the control plane has also made distribution easy to operate and debug: engineers can understand which decisions led to low availability, high latency, or poor hit rate because these decisions are made by a tracker with a consistent view of distribution state.

When workloads scale beyond the capacity of a single tracker, the detailed state is sharded across several cooperating trackers, each managing a distinct set of peers. Trackers exchange lower-fidelity views of their individual state with other trackers. Thus, each tracker has a fine-grained view of the state it manages and a coarse-grained view of the entire state. Trackers use the coarse-grained view to delegate decisions to other trackers when using peers that those trackers manage.

The second major problem faced by prior distribution systems was a lack of flexibility. At Meta, clients have vastly different resources to spare for distribution; e.g., some clients can dedicate gigabytes of memory or disk for peer-to-peer caching, while others have no resources to spare. Client have very different access patterns and scale. Finally, the objectives for distribution can differ: some clients need low latency, while others wish to reduce the load on external storage to avoid throttling or excess quota requests. The variety in client needs was one reason Meta needed many different distribution solutions; each solution was customized for a small set of use cases. To unify the disparate distribution solutions, we could not simply provide a one-size fits all solution because that would regress many clients on their key metrics.

We therefore chose to make customization a first-class design priority. Trackers implement modular interfaces for specifying different *policies* for caching and fetching data. Further, each policy is itself configurable to allow for different tradeoffs across client types and responses to changing workloads. We use trace-driven emulation to search through the space of possible customizations and find the best policies and configurations for each observed workload.

This paper describes our solution, Owl, a highly-customizable data distribution system with a centralized control plane and a decentralized data plane. Owl has been in production use at Meta for almost 2 years. Owl has scaled out rapidly (production traffic increased by almost 200x in 2021). Currently, Owl has over 10 million unique clients (binaries concurrently using the Owl library), and it downloads over 800 petabytes of data per day. Owl supports 106 unique types of clients and has customized policies for 55 of these. In production, Owl improved download latency over prior systems by a factor of 2–3 for our most important use cases, while requiring only a fraction of the resources needed by prior centralized solutions.

In summary, this paper makes the following contributions:

1. It shows that a centralized control plane need not be a barrier to scalability in peer-to-peer distribution.

2. It shows that tracker sharding and delegation retain the benefit of fine-grained management even when load grows beyond the capacity of a single tracker.

3. It shows that first-class support for flexible distribution

[1] borrowing terminology from BitTorrent

Client API function and arguments	Description
`read_blob (object, offset, length, deadline, integrityChecker, decryptor)`	Fetches all or part of an object to memory.
`read_blob_to_file (object, fd, offset, length, deadline, integrityChecker, decryptor)`	Fetches all or part of an object to a file.
`provide_file (object, fd, length)`	Allows a file to be distributed ephemerally.
`evict_file (fd)`	Evicts a file from the peer cache.

Table 1: **Owl client API**

and caching policies can provide substantial gains in efficiency and latency, especially when combined with tools that automatically search the space of possible policies for optimizations.

2 Design and Implementation

Owl has two basic components: *peers*, libraries linked into every binary that uses Owl to download data, and *trackers*, dedicated Owl services that manage the control plane for a group of peers. A physical host often has several Owl peers due to container stacking and use of Owl by the Twine container infrastructure [16]. Each tracker manages many peers: over 10 million Owl peers are currently managed by 112 trackers. Additionally, Owl has approximately 800 *superpeers*, dedicated services running the Owl library that provide extra caching or perform specialized tasks.

2.1 Peers

Owl peers provide a simple API for downloading data, shown in Table 1. Client processes call `read_blob` to fetch content from a source object, specifying a range of data to read. The object name encodes an external storage source and a unique identifier for the object within the external storage namespace. Owl currently supports 3 types of external storage. The caller can optionally specify a deadline and classes that check data integrity or decrypt provided data (discussed in Section 2.9). `read_blob` returns a reference-counted memory buffer, while `read_blob_to_file` writes the content to a file. The `provide_file` function allows peers to provide ephemeral content, as discussed in Section 2.11, and `evict_file` lets clients manage disk caches shared with Owl, as discussed in Section 2.7.

Owl peers cache data in memory and on disk. These caches may be shared with the client binary if the client does not modify downloaded data. Owl uses the caches to serve content requests from other peers. Owl policies usually prefer to fetch data from a peer rather than from an external data source, so most requests are satisfied by peer-to-peer distribution.

In the design of Owl, a key principle is that peers should be as simple as possible. This is achieved via a mechanism-policy split, where peers provide the mechanism to perform simple actions such as downloading a chunk of content from a single source, caching or evicting a chunk in memory or on disk, or providing cache data in response to a request from another peer. When downloading content, peers ask trackers to decide from where they should fetch content, how they should retry failed downloads, and even which chunks they should cache locally.

This design principle has been invaluable for operational simplicity. At Meta, the Owl team can control the deployment of its own service (i.e., trackers and superpeers); however, Owl peers are linked with client binaries and so deploy according to different schedules controlled by many other teams. The Owl team deploys code changes to trackers daily, and the team can change configuration values on trackers within seconds if necessary. In contrast, peer code changes can take months to fully deploy. By keeping peers as simple as possible, the team minimizes the need to change a widely-deployed and hard to modify part of the system.

Each peer is associated with a *bucket*, which uniquely identifies the type of the client binary with which the library is linked. The bucket provides a way to customize Owl behavior for each type of client and it lets us monitor usage, performance, and reliability for each Owl customer individually. Currently, Owl supports production traffic for 106 buckets.

2.2 Trackers

A tracker manages download state for a set of peers. Typically, peers and trackers are grouped by region (a region is several co-located data centers), with 3–4 trackers per region providing scale and redundancy. Trackers are homogeneous and multi-tenant. In general, each tracker supports all Owl buckets, and the association between peers and trackers in a region is random. However, Owl uses a separate set of trackers in each region for binary distribution to provide strict performance isolation for this sensitive workload.

Trackers associate data and peers. Downloaded objects are divided into chunks; chunk size varies by bucket with 50 MB being the most common size. For each chunk, tracker metadata specifies which peers are caching the chunk and which are downloading it. Tracker metadata also specifies the source of each peer's download (e.g., an external source or another peer). For each peer, the tracker metadata specifies the

peer's location (host, rack, region, etc.) and its cache state (the chunks in the cache, last access time, and so on). In contrast to highly-decentralized systems, Owl trackers can maintain such detailed state because trackers make all major decisions about caching and downloading chunks on behalf of peers.

As our evaluation shows, a single Owl tracker can scale to handle 1.5–2.4 TB/s of distribution traffic, depending on assumptions about cache hit rate for download requests. To achieve this scalability, we have used careful, but mostly standard, engineering practices. Trackers are implemented in C++ and use common abstractions (coroutines, reader-writer locks, and standard library containers). Trackers maintain geographically-sorted indexes to order peers and the chunks they cache by location; these indexes allow trackers to efficiently find the nearest peers caching a particular chunk of data. Geographically-sorted indexes are used frequently by location-aware selection polices. Trackers store all metadata in memory, and they rebuild their state quickly when restarted.

Peers associate with one tracker. Each peer picks a random instance from the set of available trackers and registers by sending an RPC. Peers register with a new random tracker if their association with the current tracker fails. Section 2.8 describes how peers are sharded across multiple trackers.

2.3 Superpeers

Superpeers are tasks running the Owl peer library as a standalone process (without any client). Superpeers sometimes provide specialized functionality. For example, some external storage systems use mountpoints that are not available on most hosts, so we access this storage only via superpeers that have been configured with the necessary mountpoint. To read external storage, the tracker directs such a superpeer to fetch and cache a data chunk, and it directs a reading peer to get the data from the superpeer.

Some Owl buckets need more peer resources than their clients can provide. For example, some clients are extremely memory and disk constrained and yet also require a high cache hit rate to reduce load on external storage. Superpeers can use all the resources of their hosts for caching, and so Owl uses superpeer caches to supplement peer caches for such buckets.

When used in this manner, a collection of superpeers can be viewed as a hierarchical caching layer. It is possible to craft Owl selection policies that direct all requests to superpeers and bypass fetching from other peers. Early in the project, we created one such policy to support an AI bucket that could spare no memory or disk for peer caching. However, we soon found that shared caching, discussed in Section 2.7, allowed Owl to temporarily access data buffers in use by the application to provide a decent peer-to-peer cache hit rate. Superpeers are still valuable because their additional caching resources improve the total Owl cache hit rate from the base peer-to-peer rate up to the target needed by the AI team. Currently, the Owl team discourages superpeer-only policies because there are several existing systems at Meta that provide excellent standalone caching solutions.

At the other end of the spectrum, it is also possible to craft Owl selection policies that do not use superpeers at all. For buckets with very large working sets and large numbers of clients, the additional cache resources of superpeers make little difference in overall cache hit rate. In practice, though, Owl selection policies for these types of buckets still use superpeers as a last level of retry if a direct fetch from storage by the peer fails. We have found this to be useful in handling rare corner cases such as particular peers being in a bad state where they cannot fetch from external storage. Because peers run on heterogeneous hosts not owned by the Owl team, they can be less stable than superpeers. The superpeer layer thus still plays a role in improving overall data availability.

Superpeers have occasionally been quite useful in mitigating production issues that lead to a poor cache hit rate. In such scenarios, we have quickly stood up a large number of superpeers in a region to provide temporary caching that restores the desired hit rate until we are able to deploy a fix for the underlying problem.

We originally implemented superpeers as a sharded service built on a standard caching library [3]. This approach proved to be insufficiently flexible; it was difficult to customize superpeer cache behavior for each bucket. Later, we rewrote superpeers to use the Owl peer library, which let us customize superpeers via tracker policies and which also provided the simplicity of code reuse for peers and superpeers. From the point of view of a peer, there is no distinction between fetching a data chunk from a superpeer or another peer.

2.4 Tracker-Peer Communication

Peers register with a tracker by sending an RPC with their bucket and location. Trackers customize behavior by bucket; e.g., tracker configuration parameters assign specific download and caching policies to each bucket. When registering, peers select trackers randomly within a geographic scope, and the association between a tracker and peer persists until a peer terminates or cannot communicate with the tracker. On failure to communicate with a tracker, peers re-register with a randomly-chosen tracker.

To download data, an Owl peer first makes an RPC to the tracker specifying the object to be downloaded and the range of data to read. The tracker returns the chunk size (determined by the bucket configuration). It initializes a state machine to track the download of each chunk that has data in the specified range. If the download later fails or times out, the tracker cancels all per-chunk state machines for the download. Otherwise, each chunk is handled independently.

Peers download chunks in parallel. Download concurrency is limited by the per-bucket configuration, which specifies both the maximum number of chunks each peer can download in parallel and a maximum number of chunks that can be read

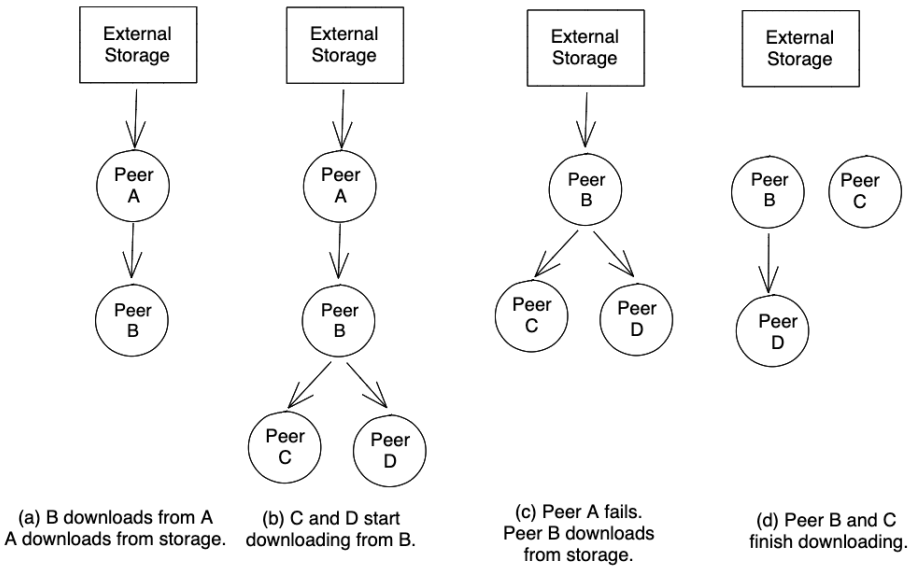


Figure 1: **Ephemeral distribution tree.** This figure shows how a tracker uses a per-chunk ephemeral distribution tree to track which peers cache a chunk and which are downloading the chunk.

in parallel for each individual `read_blob` request.

For each chunk, the peer first checks if the chunk is in its cache already. If so, it sends an RPC to the tracker, which terminates the per-chunk download state machine and updates the access time in the peer metadata for LRU and similar eviction policies. Otherwise, the peer sends a `getSource` RPC to the tracker that asks how it should get the data. The tracker can respond with a peer from which data can be obtained. Alternatively, the tracker may specify an external data source from which the peer should fetch content directly. The `getSource` response specifies whether the peer should cache the downloaded chunks and lists chunks that should be evicted from the peer cache to make room. The tracker updates its peer and chunk metadata, along with the per-chunk download state machine to reflect its decision.

The peer next attempts to obtain the data in the manner specified by the tracker, and it informs the tracker of the result. On success, the tracker terminates the state machine. On failure, the tracker makes a new decision based on its current metadata. Based on per-bucket policies and the failure type, the tracker may specify a new source (or possibly retry the same source in rare cases), or it may tell the peer to give up (e.g., because a maximum number of retries has been exceeded or it believes the chunk is not available from any source).

Prior systems such as BitTorrent [7] provide a list of candidate sources to a peer and let peers handle retries transparently. Owl's approach of involving trackers in retry decisions has several advantages. First, trackers can pick a new source based on the latest state about which peers cache the chunk and current peer load. In contrast, the peers included in BitTorrent's initial list can be stale when retries are needed. Second, Owl trackers maintain very detailed state about which peers are fetching from others. This allows trackers to enforce precise caps on the maximum number of inflows and outflows per peer, and it allows trackers to make more informed selection decisions. Finally, this detailed state gives operators a complete picture of Owl download state, making it easier to determine why downloads may be slow or failing.

The peer simply follows the tracker's instructions at each step. On retry, if a prior step returned partial content before failing, the peer resumes fetching from a new source after the last byte it received (so failures do not lead to excess data being transmitted).

If a chunk download fails (because the tracker tells the peer to give up) or the download time exceeds a deadline specified by the client, the peer cancels all remaining chunk downloads and fails the `read_blob` request. Otherwise, the peer returns the requested data either via a memory buffer or by writing to a specified disk file. The tracker also sets a timeout for each chunk download; it terminates a download and cleans up download state if the peer does not respond within this time.

2.5 Ephemeral Distribution Trees

The *ephemeral distribution tree* is the core abstraction used by the tracker to manage per-chunk download state. The root of a tree is an external data source or a peer that caches the chunk. Directed edges indicate which peers are actively downloading the chunk from others; e.g., in Figure 1(a), peer A

is downloading the chunk from external storage, and peer B is downloading the chunk from peer A. While prior distribution systems have commonly used trees to efficiently distribute data, Owl's trees are particularly ephemeral in that each chunk of data has its own forest of trees, and nodes remain in a tree only while they are downloading a particular chunk or providing a chunk to another peer.

In the data plane, chunks are streamed from the root to leaves; i.e., bytes along each edge are sent in order, followed by a per-chunk checksum used to verify integrity. Each peer forwards data to its children as soon as it receives new bytes. With large chunks, this design means that tree depth does not strongly affect latency. Leaf-nodes see only the first-byte latency of additional communication hops, which is often quite small within a data center or region. In Figure 1(b), when peers C and D request the chunk; the tracker tells them to get it from B, which is still receiving data. Peer B first sends its cached bytes and then forwards additional chunk bytes as it receives them.

When a peer reports a failure fetching data, the tracker removes the edge connecting the peer to its parent. If the tracker chooses a new source, it creates an edge from the peer to that source. Thus, the entire subtree rooted at the peer reporting the failure is moved to a have a new parent in the tree. When choosing a new peer, the tracker avoids creating download cycles; i.e., it will not designate a descendent of a peer as a new source for that peer. Tree repair minimally impacts downstream nodes because Owl resumes a new download after the last byte fetched from the previous attempt. In Figure 1(c), peer A fails, and the tracker tells peer B to fetch the remaining bytes from external storage. Peers C and D are oblivious to this change since they continue to download from B.

When a peer reports a successful download, the edge connecting it to its parent is removed. The tracker adds the peer to the list of nodes that have the chunk fully cached if the tracker asked the peer to retain the chunk. Since chunks may be cached at multiple peers, the download state for a chunk is a forest of ephemeral distribution trees rooted at multiple such peers and/or the external data source. In Figure 1(d), peers B and C have downloaded and cached the chunk, while peer D is still downloading bytes from B. Thus, we have two ephemeral distribution trees in the forest; a new peer that requests the chunk may be directed to any of these peers or to external storage, depending on the selection policy for the bucket.

At first glance, it might seem surprising that Owl often prefers to download chunks from peers that have partially downloaded a chunk in preference to peers that have the chunk fully cached. However, selecting a peer that has partially downloaded a chunk has little latency cost. The peer immediately starts streaming out the bytes it has already downloaded and sends remaining bytes out as soon as they arrive. Network locality and quick scale-out of hot contents are bigger concerns in practice. For instance, many peers in a rack often request a chunk at the same time. Most Owl policies are location-aware and build a tree so that a single peer downloads data from outside the rack and other peers in the same rack get the chunk from that peer or one of its children. Similarly, if many peers in a data center request a chunk at the same time, typically only one peer fetches the chunk from outside the data center. Allowing peers to fetch from other peers that are still downloading the chunk is essential to achieving network locality for hot content.

2.6 Selection Policies

Each bucket has a selection policy that the tracker executes on each `getSource` request. The selection policy considers the result of all prior attempts by a peer to fetch a chunk, as well as per-chunk state that includes the set of caching peers and ephemeral distribution trees. The result of the selection policy often directs a peer to fetch the chunk from another peer or an external data source; these decisions add a new edge to an ephemeral distribution tree. The policy is implemented as a class inheriting from an abstract interface; each policy class has a considerable number of parameters that can be further customized via configuration [15].

A selection policy may use a superpeer to assist in the download. The tracker directs the superpeer to fetch the chunk from an external source, and it directs the requesting peer to get the chunk from the superpeer. This creates a 2-edge distribution tree. Usually, the tracker will have the superpeer cache the chunk so other peers can fetch the same chunk without reading from external storage; this is especially useful when many chunk requests arrive within a short time window.

The *location-aware* policy is the default selection policy. This policy selects the nearest peer that caches or is downloading the chunk, subject to per-peer constraints on maximum fanout and bandwidth usage. Distance is determined by network topology; peers on the same host are preferred over peers in the same rack, which are, in turn, preferred over peers in the same network cluster, etc. To make location-aware selections quickly, the tracker maintains a topological sort over all peers caching or downloading a chunk. A selection policy also specifies the number and type of retries. By default, the location-aware policy tries up to 5 peers, then tries to fetch the chunk via a superpeer, then tries to fetch the chunk directly from a source before giving up. The policy also has unique handling for specific errors, such as external source throttling.

Another common policy is the *hot-cold* policy, which refines the location-aware policy by using superpeers for hot data. If no peer can provide a chunk from its cache, this policy reads data from an external source via superpeers if the chunk is hot or directly from the source if the chunk is cold. Hotness is determined by examining the number of chunk reads within a recent time window. The policy improves hit rate in superpeer caches for buckets that have a mix of hot and cold content.

Other policies implement load balancing; e.g., spreading

```
1  Decision selectSourceForData(
2      const ChunkMetadata& MD,
3      std::shared_ptr<PeerMetadata> requester,
4      const ChunkStatus& stat,
5      const ShardedChunksMap& shardedChunks,
6      const DownloadContext& context) override {
7
8    if (hasDirectFetchFailed(stat)) {
9      return Decision{GIVE_UP, nullptr};
10   }
11
12   if (cannotFindSuperpeer() ||
13       noMoreSuperpeerAttempts(stat) ||
14       noMoreAttempts(stat)) {
15     return Decision{DIRECT_FETCH, nullptr};
16   }
17
18   if (noMorePeerAttempts(stat)) {
19     return Decision{SUPERPEER_FETCH, nullptr};
20   }
21
22   peer = selectPeer(MD, requester, stat, context);
23   if (peer) {
24     return Decision{PEER_FETCH, peer};
25   }
26
27   delegation = findDelegation(shardedChunks);
28   if (delegation) {
29     return Decision{DELEGATED_FETCH, delegation)};
30   }
31
32   return Decision{SUPERPEER_FETCH, nullptr};
33 }
```

Figure 2: Pseudocode: Location-Aware Selection Policy

downloads from sources evenly. Still others always fetch via superpeers, select random peers, and direct whether and how chunks should be fetched from out-of-region peers.

To illustrate how policies are written, Figure 2 shows pseudocode for Owl's location-aware selection policy. Each policy is implemented by overriding a C++ base class; in this case we show the `selectSourceForData` method, which is used to determine how and from where a peer should fetch data on each chunk download attempt. The method's inputs are: chunk metadata that includes a topologically sorted index of all peers and superpeers caching the chunk, metadata describing the peer requesting the data that incudes its location info, a status object containing all prior attempts to fetch the chunk for this download and their results, a list of other trackers that have the chunk available for delegation, and bucket-specific context about the chunk.

Policy implementations are usually a series of simple rules. The location-aware policy first calls a helper function (line 22) to select the nearest peer or superpeer caching or downloading the chunk, as long as such a peer is healthy (no recent failures reported) and would not exceed limits on number of downloads, network bandwidth, etc. The helper function considers past attempts and only tries each source once.

If the tracker has no more locally-managed peers or superpeers caching the chunk, it tries to find a delegation for the chunk from a peer tracker (line 27). If this fails, the policy asks a superpeer to fetch the chunk from an external source, cache it, and provide it to the requester (line 32).

The policy has configurable limits on the number of peer and superpeer attempts. If there are no more peer attempts allowed, the next retry asks a superpeer to fetch the chunk from an external source (lines 18–19). If there are no more superpeer attempts left or the policy has attempted to find a free superpeer and failed, the peer is asked to fetch the data from the external source directly (lines 12–15). If this direct fetch fails, the policy gives up (lines 8–9).

2.7 Caching policies

Per-bucket caching polices determine how peers cache data. Peers may cache data in memory or on disk, with some buckets using both types of cache. Cache size is configurable; the default memory cache size is 1 GB but size varies widely across buckets, depending on memory constraints and desired cache hit rates.

Some buckets use a *shared* peer cache, in which a single copy of data is shared read-only between the client application and Owl distribution. For in-memory caching, Owl returns a reference-counted buffer from `read_blob`. The buffer remains in the cache until the client releases the reference. For example, one memory-constrained client type reads AI models using a shared cache. While the client has no memory to spare, it retains data read for several seconds while it transforms the model chunk into a different format. By sharing the buffer with the client, Owl can satisfy many peer-to-peer download requests during this time. This sharing is essentially free because the data would reside in memory for the transformation anyway. This particular bucket needed a good hit rate to avoid overloading its external storage. Shared caching got us most of the way there, and we used superpeer cache capacity to further improve the hit rate to meet the bucket's requirement.

Buckets with disk caching often use shared caching. Downloaded objects are written to files with Owl retaining an open file handle so that it can serve cached file content to peers. The client controls when files are garbage collected by calling `evict_file` in Table 1. Owl also provides an interface that watches downloaded files and calls `evict_file` on the client's behalf if the file is deleted. In lieu of controlling eviction explicitly, some clients provide a *TTL* (time-to-live) that specifies how long downloaded data should be cached before eviction.

For private (non-shared) caching, Owl trackers manage peer caches. On each `getSource` request, the caching policy determines whether the peer should cache the requested chunk and which chunks to evict to make room in the cache. Peers specify their current cache state when registering with a new tracker so management persists across tracker failures.

The default caching policy is LRU (least recently used). Another popular policy, used for shared caching, never evicts chunks because the eviction is done explicitly by each peer.

Many clients that need good peer-to-peer cache hit rates use a *least rare* policy that prefers to evict chunks cached on more peers over chunks cached on fewer peers. A hybrid policy uses least-rare eviction for hot chunks and LRU eviction for cold chunks. Owl also supports random chunk eviction, which often has good properties for hot data [18].

2.8 Tracker sharding

For the first year of operation, Owl used a single tracker per region, with hot spares providing primary-backup fault tolerance. The simplicity of a single tracker allowed us to start serving production traffic 3 months after the start of the project. However, we knew that our workload would eventually exceed the capacity of a single tracker. Thus, we added the capability to shard peers across multiple trackers.

With sharding, trackers have equivalent responsibilities. A sharded tracker maintains the complete peer state for a given set of peers, but per-chunk and per-download state is split across the shards. Peers and superpeers register with random trackers.

Sharded trackers periodically exchange the set of chunks cached by at least one peer or superpeer that they manage. Trackers normally send incremental updates once a second with additions to and removals from this set. However, a receiving tracker may request a full snapshot when needed; e.g., because it just restarted or it missed an incremental update. Thus, each tracker has a coarse-grained and slightly stale view of the global distribution state that maps chunks to trackers rather than to specific peers.

Selection policies can decide to fetch a chunk from another sharded tracker; typically, this happens when the chunk is not cached on any peer managed by the local tracker and another tracker has reported that it has the chunk. The tracker running the selection policy sends a *delegation* request to the other tracker. In turn, that tracker selects and returns a peer caching or downloading the chunk. The delegation request fails if no such peer exists.

On successful delegation, each tracker updates state for the peer it manages. The `getSource` response simply specifies the endpoint of the delegated peer, so peers are oblivious to delegation. When the downloading peer reports success or failure, its tracker forwards the report to the delegating tracker and both trackers update their individual state accordingly.

On receiving a successful delegation response, a tracker starts a new ephemeral distribution tree. The root of a tree is a *delegated peer*, which indicates that the peer is managed by another tracker. The tracker grows the tree as other peers request the chunk, since selection policies commonly prefer to fetch from a locally-managed peer over a delegated one.

The ephemeral distribution tree for a chunk is now partitioned across multiple trackers with a node in the tree of one tracker serving as the root of a subtree in another tracker. In order to prevent cycles in this partitioned tree, a tracker will not provide any peer in a tree rooted at a delegated peer in response to a delegation request.

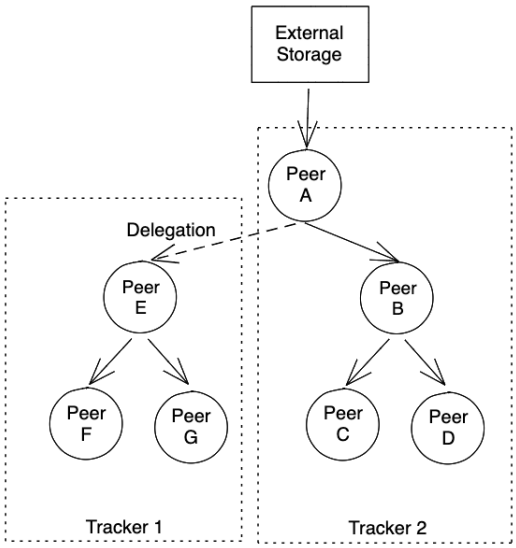


Figure 3: **Delegation with 2 sharded trackers.** Peer E fetches a chunk from a peer managed by another tracker to reduce load on external storage.

Figure 3 shows an ephemeral distribution tree sharded between 2 trackers. Tracker 2 initially receives a `getSource` request from peer A and instructs peer A to read the chunk from external storage. At this point, tracker 2 starts advertising that it has the chunk to other sharded trackers. Next, tracker 1 receives a `getSource` request from peer E. It does not have the chunk on any of its peers, but it knows that tracker 2 has advertised the chunk. Tracker 1 sends a delegation request to tracker 2, which selects and returns peer A. Tracker 1 tells peer E to fetch the chunk from peer A. When tracker 1 receives subsequent `getSource` requests from peers F and G, the bucket's selection policy prefers locally-managed peers, and so these peers are directed to fetch from peer E. As this example shows, delegation improves cache hit rate for sharded trackers. Without delegation, both peers A and E would fetch from external storage. With delegation, there is only a single fetch by peer A, which achieves the same overall cache hit rate that would have been achieved without sharding.

Some data sources accessed by Owl are regional. In these cases, when an out-of-region peer requests a chunk that is not cached by another peer, selection policies use delegation to ask an in-region tracker to have one of its superpeers read the chunk. The requesting peer is directed to that superpeer for the data. Selection policies consider cross-region latency to find the closest location from which to read data.

2.9 Security and integrity

All communication between Owl components in encrypted, and all RPCs are checked against access control lists. Many data sources read by Owl encrypt data at rest, so chunks in Owl caches are often encrypted. Owl clients can provide decryption functions to read decrypted data. For shared caching, Owl must cache and share unencrypted data with clients (because that is how they consume the data). In this case, Owl decrypts each chunk when writing it to the cache and reencrypts it to share it with another peer.

Owl generates an internal checksum when reading chunks from external sources, passes the checksum with the chunk data, and validates chunks with the checksum before returning them to clients. Many Owl clients generate end-to-end hashes when writing to external storage. These clients can optionally provide Owl an integrity checker class containing these hashes and the hash calculation function to validate that the data being read is the same as what they originally wrote. Owl calls the integrity checker as data is being written to a disk or a memory buffer. It fails the download if the integrity checker reports that the calculated hash does not match the write hash.

2.10 Virtual superpeers

One of our original design principles for Owl was that peers should not fetch content that their clients do not read. This led to high network efficiency and made it easier to convince users to adopt Owl since their clients would not be doing work for other services.

However, one recent bucket demonstrated a drawback with this approach. For this bucket, reducing load on external storage is crucial; if data is read too fast, the external storage system throttles readers and performance degrades rapidly. Periodically, a new search index is generated and distributed, which each client then reads at a random time over the next few hours. The first client reads the index directly from external storage, but its memory cache fits only a few chunks. The next client reads those chunks from the first client, and it reads the remainder of the index from external storage. As more clients download the index, their collective caches are eventually sufficient to hold all the data (especially since we use the least-rare caching policy to maximize hit rate for the bucket). However the clients together read many extra chunks from external storage until the index is fully cached, and this causes the external storage system to throttle readers.

To solve this problem, we added a new Owl abstraction called a *virtual superpeer*. If a bucket is configured with a virtual superpeer, the tracker divides each peer's cache into a normal portion and a portion reserved for the virtual superpeer. The tracker aggregates the virtual superpeer portions and manages the collection in the same way that it would manage a superpeer dedicated solely to the bucket. When the first client reads the index, the bucket selection policy routes the ephemeral distribution tree for each chunk through the virtual superpeer. The tracker uses the per-bucket policy to select one peer to fetch the chunk from external storage and cache it; the tracker also selects chunks to evict from the virtual superpeer portion of that peer's cache, if necessary. The requesting peer streams each chunk from the peer that fetched it from the external source. After the index is loaded by one peer, the next peer to fetch the index finds all chunks in the virtual superpeer cache. Thus, Owl makes no additional reads to external storage, and it achieves a high cache hit rate.

The benefit of virtual superpeers over non-virtual (physical) superpeers is that virtual superpeers use spare memory capacity on peers rather than dedicated machines. The bucket described in this section would require approximately 640 physical superpeers to achieve the same cache hit rate as Owl achieves with virtual superpeers. Another bucket that we are currently onboarding would require approximately 10,000 physical superpeers; we are avoiding this cost by leveraging spare peer memory via virtual superpeers.

Virtual superpeers are a tracker-only concept; peers are unaware of the abstraction because they simply follow tracker instructions for where to fetch data and which chunks to cache. Further, the abstraction is implemented almost entirely via Owl selection and caching policies (we added a few hundred lines of tracker code to implement cache partitioning and eliminate double-buffering). Overall, virtual superpeers demonstrate the flexibility of Owl policies: we were able to implement a substantial change not envisioned in the original Owl design primarily by writing new policies.

2.11 Ephemeral data sources

Owl was originally designed to download content from external storage. However, several clients wanted to use Owl to distribute content produced by instances of their service directly to other instances, bypassing storage entirely. For AI models and search indexes that have diminishing value over time, durable storage provides little benefit. Yet, the resources used to read and write large data objects to distributed storage can be significant. We modified Owl to support these use cases by adding *ephemeral data sources*.

An ephemeral data source is simply a peer that promises to supply specific content when requested. The client calls `provide_file` in Table 1 to specify a file containing content for a given unique identifier. In turn, the peer tells the tracker that this content is now in its cache. When other peers requests chunks from this content, the tracker builds ephemeral distribution trees rooted at the providing peer to distribute the chunks. The tracker also advertises and provided content to other sharded trackers, which makes the content available via delegation.

Owl guarantees that the data will be provided only as long as an ephemeral data source provides the data. It caches ephemeral content on peers and superpeers as normal, and

it falls back to the peer(s) providing the data as a last resort. Ephemeral data sources must re-register with a new tracker if their connection with the current tracker fails; they send heartbeats every second to their trackers to proactively detect failures and re-register quickly. A client may stop providing content by calling `evict_file`.

2.12 Fault tolerance

Tracker sharding allows Owl to tolerate tracker faults. When a peer detects that its tracker has failed, it re-registers to use a new tracker. Trackers have only soft state, and a new tracker learns a peer's existing cache state as part of registration. We regularly test failover by continuously deploying tracker code each workday, during which trackers are sequentially killed and restarted. We occasionally experiment by killing and restarting all sharded trackers simultaneously to ensure that performance does not drop below SLA bounds when all trackers restart. This has proved enlightening; e.g., we added peer re-registration when we noticed that SLA bounds had been violated during one such trial.

The RPC routing layer at Meta (not part of Owl) load balances requests among sharded trackers at the granularity of each new chunk download. However, peer associations with trackers are typically long-lived, as rebalancing does not need to be done often in steady state. In contrast, Owl load balances superpeers among trackers itself because the number of superpeers per tracker is small and we want to maintain a tighter balance than the RPC router layer provides.

Trackers detect peer failures when a peer reports that it cannot reach a peer from which it is trying to get data. Peers are marked down (and not used to serve further requests) after a configurable number of consecutive failures. Peers are marked up again when they re-register with a tracker. The Owl library explicitly deregisters on shutdown, but many peers don't shut down cleanly, in which case there is no deregistration.

Generally, we do not allow peers to fail over and use trackers outside their region. Experiences with other systems left us concerned about cascading failures in which a failure in one region causes out-of-region requests to overload services in other regions. We use a separate set of global trackers for buckets that require peers to contact out-of-region trackers.

2.13 Emulation and customization

Over time, Owl has become more customizable as we have added new policies and enabled different behavior via configuration within each policy. This flexibility often makes it difficult to determine the best set of policies for each bucket.

The Owl team writes all policies and helps Owl users pick the best policy for their needs. If the team identifies a specific need not covered by an existing policy, we write a new policy; the development of the virtual superpeer policy, described in Section 2.10, is a good example of this process.

Choosing the best policy is difficult. Many engineers who wish to use Owl do not understand their service traffic patterns well. In some cases, it is not clear whether their workload would benefit from peer-to-peer distribution. In other cases, it is difficult to choose the best set of policies or explain specific configuration tradeoffs; e.g., how much additional cache hit rate can the bucket expect for each additional gigabyte of peer memory used? For existing users of Owl, traffic patterns and distribution goals change over time (e.g., a service can spare less memory or require better cache hit rate to reduce external storage load). Thus, initial policy choices often need to be tuned to keep pace with client changes. As the number of buckets using Owl grew, it became infeasible for the Owl team to manually choose and tune policies for each unique workload.

Owl uses offline, trace-driven emulation to guide policy choices. On a per-bucket basis, Owl can be configured to log basic information about each client request to a database; e.g., the timestamp of the request, the object, and data range read. When we onboard a new bucket that reads data from an external source, we first use an *evaluation mode* policy that always instructs the peer to fetch from the external source and not cache data. The peer thus performs the same actions it would perform without Owl except that each request is routed through a tracker for logging. For existing buckets, logging is always enabled.

An Owl emulator runs our actual tracker service with mock peers and superpeers that generate traffic and service requests. The emulator is event-driven and uses a virtual clock to determine when events occur. Mock peers register, deregister, and generate requests at the times recorded in the production traces. The emulator adds configurable network and storage delays, and can simulate different error profiles. Because we run the actual tracker code, we can emulate any Owl policy or configuration. The emulator reports key statistics such as overall cache hit rate, load on external storage, and tracker CPU usage.

To find the best policy, we compare statistics from multiple emulation runs with the same trace and different policy/configuration settings. As the setting space is quite large, we use random-restart hill climbing [10] to search for the best choice for each bucket. Currently, we run the emulator weekly on existing Owl buckets, and we use the emulator to evaluate all new buckets during onboarding.

3 Evaluation

Our evaluation answers the following questions:

1. How well does Owl provide hot content distribution in production?

2. How does Owl compare to other centralized and decentralized distribution solutions?

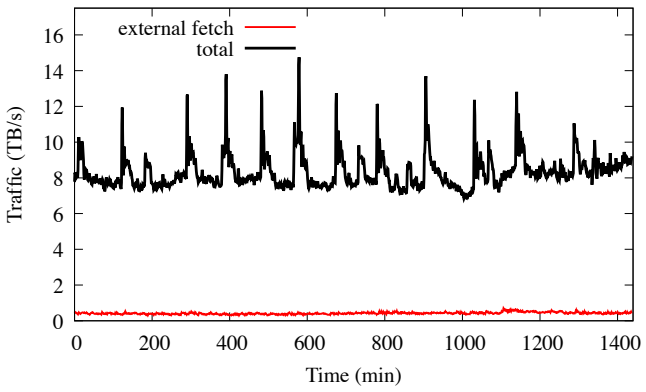


Figure 4: **Owl traffic over a 24 hour period.** The top line shows total bytes downloaded by all clients averaged every minute. The bottom line shows the total bytes read from external storage. The difference between the two lines is the reduction in external storage load due to Owl peer-to-peer caching and distribution.

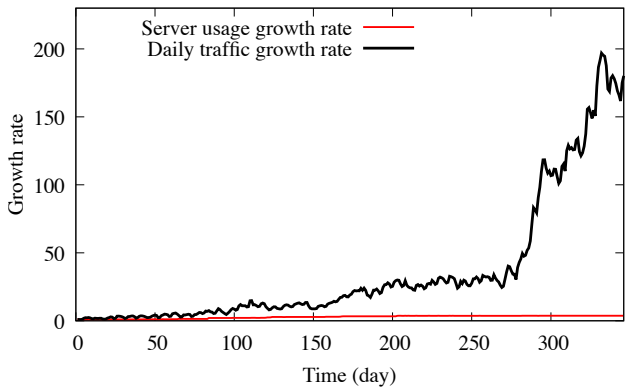


Figure 5: **2021 growth in traffic and server usage.** The top line shows Owl's daily 2021 traffic load relative to the load at the beginning of the year, and the bottom line shows the number of servers (trackers and superpeers) used in production relative to the number used at the beginning of the year.

3. How much benefit does Owl realize from delegation?
4. What are the benefits of flexible distribution policies?
5. How well does Owl scale and how many peer resources does it require?

3.1 Reducing load on external storage

Figure 4 shows a recent (and typical) 24 hours of Owl production traffic, aggregated by minute. The top line is the amount of data read by clients; this is the load they would impose on external storage without Owl. The bottom line shows the load on external storage with Owl. During the 24 hours, Owl clients read 717 PB of data, yet only 36.5 PB was read from external storage, for a cache hit rate of 94.9%.

The cumulative read rate across all Owl clients varies from a minimum of 6.84 TB/s to a maximum of 14.75 TB/s. Figure 4 shows that peer-to-peer distribution and caching hides the client load spikes almost entirely from external storage; in fact, the load on external storage never exceeds 0.72 TB/s.

A CDN or hierarchical caching could also reduce the load on external storage, as in a recently reported study of CacheLib [3]. In that study, each caching node could sustain a maximum data rate of approximately 640 MB/s. Thus, even assuming perfect load distribution, it would require over 23,000 caching nodes to handle Owl's peak client request rate for the reported period, which is more that 200 times the number of current Owl trackers (112).

Figure 5 shows the scalability benefit of Owl's decentralized data plane by comparing the relative growth in production traffic and servers (trackers and superpeers) in 2021. While Owl's peak traffic is almost 200 times greater than traffic at the beginning of the year, server usage has grown by less than a factor of 4.

When Owl replaced hierarchical caching solutions at Meta, we also saw latency speedups from 50% to 100% for several large buckets due to better network locality and elimination of throttling errors.

3.2 Benefits of delegation

Figure 6 shows the number of successful and failed delegation requests for all Owl trackers over the same 24 hour period. 97.5% of delegation requests are successful; they return a peer that provides the requested content. The primary reason why delegation requests fail is because a sharded tracker's list of cached objects is stale. The low rate of delegation failure indicates that a 1 second update interval is sufficient for the majority of our workloads. We verified this by reducing the delegation interval to 250 ms in one region for 24 hours. Owl's largest bucket saw only a 1% improvement in cache miss rate, and overall cache miss rate did not improve within experimental error.

Delegation provides 10.1% of the total data read by Owl in the 24 hour period. In other words, without delegation, the Owl cache hit rate would decrease from 94.9% to 85.4% (increasing the miss rate by nearly a factor of 3). Delegation is thus an essential factor in providing good download efficiency with sharded trackers.

3.3 Comparison with prior systems

We next compare Owl with the two peer-to-peer distribution systems it replaced at Meta. The first such system was a location-aware implementation of BitTorrent. We configured roughly half the hosts in one region to download binaries

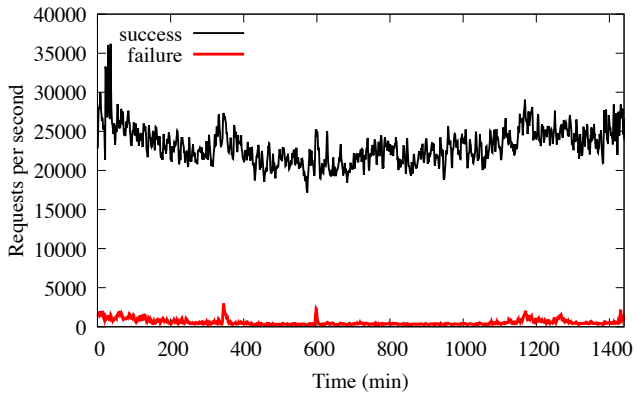


Figure 6: **Delegation success rate over a 24 hour period.** The top line shows the number of successful delegation requests, and the bottom line shows the number of unsuccessful delegation requests.

for provisioning via Owl; the remaining half used BitTorrent. Both systems had identically-sized disk caches. During a 24 hour experiment, these hosts downloaded 17.6 million binaries with a median size of approximately 300 MB. Table 2 compares results for the two systems. Owl is significantly faster than BitTorrent, almost doubling the median per-client download throughput and more than quintupling the p95 throughput. Because the client writes downloaded binaries to local storage, the maximum throughput of Owl was often capped by the available write bandwidth of local media on each host; in contrast, BitTorrent rarely reached the storage bandwidth limit. Additionally, Owl reduces the load on external storage by 42% due to its higher cache hit rate (99.21% for Owl and 98.64% for BitTorrent). Both Owl and BitTorrent provided 4 9's of availability. BitTorrent had slightly higher availability due to allowing more retries with additional backoffs; we later adjusted Owl's retry policies for this bucket to more closely match BitTorrent's policy.

The second prior download system at Meta, StaticTree, used a relatively-static distribution tree constructed via consistent hashing. Each chunk has one primary cacher that fetches the chunk from storage and caches it. The primary cacher is determined by hashing the unique chunk id and selecting a host from a membership list stored in Zookeeper [9]. Each tree level corresponds to a location type (e.g., region, data center, rack, etc.) with the node at each level and location responsible for a chunk again selected via consistent hashing. Secondary nodes at each level provide fault tolerance.

Table 3 compares important download metrics for Owl and StaticTree. Experiments ran for 1–7 days and consisted of millions of production requests to both systems. Both systems use identically-sized memory caches. Owl provides 4 9's of availability in 3 of the 5 experiments and 3 9's in the remaining experiment. StaticTree provides substantially lower availability because of the time needed to detect and route around failed nodes in the tree, as well as the need to remove failed nodes from the membership list in Zookeeper. In contrast, with Owl, ephemeral distribution trees let trackers avoid using a peer immediately for new chunk downloads as soon as that peer is suspected of being unhealthy or slow.

Compared to StaticTree, Owl improves p50 download latency by an average of 55% and p99 latency by 32% across the five experiments. While Owl's latency improvement comes partially from better failure handling, the improved latency also results from the tracker dynamically picking the best data source for a chunk on each *getSource* request. Trackers improve latency by considering load on peers and network locality based on detailed peer and chunk state.

Improving per-chunk download latency often translates into even greater improvements for application-level metrics. Table 4 compares the average time to load six different types of AI models in production via Owl and StaticTree. Owl speeds up model loading time from 1.44x to 3.48x, for an average speedup of 2.92.

Cache hit rates are roughly equivalent for the two systems (StaticTree provides better cache hit rate in 3 out of 5 experiments, but Owl's cache hit rate improvement in Bucket D is by far the most substantial). We also examined network locality for peer-to-peer data transfers between the two systems; we found locality to be roughly the same as both systems optimize for this metric.

3.4 Optimization results

Owl currently has 106 buckets that use 55 distinct policies and configurations. We use the Owl emulator to regularly search for potential policy improvements. Table 5 shows the optimizations that we found in the previous month. We report savings in either peak or total storage usage over 24 hours that we achieved by modifying bucket policies in production. All of these buckets were seeing throttling from external storage at the time, so reducing storage usage was an important goal.

For the first two buckets, emulation lets us inform bucket owners how much improvement in cache hit rate they could expect from allocating more peer memory to Owl caching. Because these clients had memory to spare, we were able to achieve a substantial reduction in peak load. For the remaining three buckets, we achieved better cache hit rate without the need for any additional peer resources simply by changing the policies used by the tracker to manage each bucket.

3.5 Overheads

We measured Owl overhead on peers by profiling one thousand hosts during production usage. Owl's CPU overhead is only 0.05% on 26-core Intel Cooper Lake processors. Owl allocates memory for data caches and for network buffers; both uses of memory are configurable and controlled by the per-bucket policy depending on the client's tradeoff between

	Availability	Cache hit rate	Per-host throughput (MB/s)		Latency (s)	
			p50	p95	p50	p99
Owl	99.994%	99.21%	130.1	20.17	2	132
BitTorrent	99.996%	98.64%	66.9	3.89	4	255

Table 2: **Comparing download metrics for Owl and BitTorrent.** Both systems are used side-by-side to download binaries in one region for 24 hours. We compare the percentage of successful downloads (availability), the reduction in load on external storage (cache hit rate), the median and 95th percentile throughput (download rate), and the median and 99th percentile download latency for five different buckets.

Bucket	Experiment duration	Downloaded bytes	System	Availability	Cache hit rate	Latency (s)	
						Average	p99
A	7 days	14 PB	Owl	99.99%	85%	46.7	47.9
			StaticTree	99.60%	86%	72.2	78.2
B	1 day	30 PB	Owl	99.99%	99.34%	48.8	107.5
			StaticTree	99.91%	99.50%	51.48	122.8
C	1 day	1 PB	Owl	99.99%	69.47%	114.7	507.9
			StaticTree	99.99%	72.52%	180.5	630.8
D	7 days	22 PB	Owl	99.91%	92.70%	44.8	119.0
			StaticTree	99.83%	81.87%	99.8	128.4
E	7 days	50 PB	Owl	99.96%	99.63%	8.5	69.1
			StaticTree	99.95%	99.47%	13.3	112.1

Table 3: **Comparing download metrics for Owl and StaticTree.** Both systems are used side-by-side in production. We compare the percentage of successful downloads (availability), the reduction in load on external storage (cache hit rate), and the median and 99th percentile download latency for four different buckets.

Model	Loading Latency (sec.)		Speedup
	Owl	StaticTree	
A	31	97	3.13
B	138	199	1.44
C	78	264	3.38
D	75	261	3.48
E	82	282	3.44
F	137	465	3.39

Table 4: **Latency improvement in AI model loading** Each row compares the average loading time using StaticTree with the average loading time using Owl for a different bucket.

performance and resource usage. Outside of these two uses, Owl uses less than 0.01% of RSS (resident set size) memory on hosts with 64 GB memory. For comparison, StaticTree uses 0.15% CPU and 0.03% memory for roughly the same workload, which is 3x the resources used by Owl.

To verify scalability, we ran a load test with Owl trackers running on hosts with 64 GB memory, a 26-core Intel Cooper Lake processor, and a 25 Gb/s NIC. Our load tests showed that each such Owl tracker can support 2.4 TB/s client traffic when the Owl cache hit rate is 99%, or 1.5 TB/s client traffic when the Owl cache hit rate is 70%. The trackers are CPU-bound at these traffic levels. As an additional confirmation of being CPU-bound, over 30 days of operation, tracker memory (RSS) stayed at 11% or less on 64 GB hosts, while CPU usage spiked up to a maximum of 37%. In practice, Owl uses many more trackers than these numbers would indicate to provide redundancy for failures, regional failure isolation, and performance isolation among critical buckets.

4 Related work

BitTorrent is the most widely-recognized solution for peer-to-peer data distribution. Classic BitTorrent [7] is highly-decentralized; the trackers simply help peers find each other. Trackers originally returned a random list of peers containing desired data, but later BitTorrent implementations introduced refinements. For instance, the BitTorrent version at Meta sorts peers by location before returning the list. Many recent BitTorrent versions replace trackers with a decentralized distributed hash table [14] for so-called trackerless torrent.

Recently, peer-to-peer distribution has been used to provision containers and virtual machines in public and private clouds. Some implementations have used BitTorrent directly [5]. Uber's Kraken [1] uses a BitTorrent-like architecture to provision containers. Kraken uses trackers that returned an ordered list of candidate peers for downloading data, and it uses dedicated seeders to read from external storage. Alibaba's Dragonfly [2] also provides peer-to-peer container provisioning. Dragonfly's SuperNodes combine tracker and

Bucket	Metric	Before	After	Optimization
A	Peak storage usage	124 GB/s	54 GB/s	Increase peer cache size from 0.2 GB to 4 GB
B	Peak storage usage	63 GB/s	27 GB/s	Increase peer cache size from 0.2 GB to 4 GB
C	Peak storage usage	95 GB/s	18 GB/s	Change selection policy from location-aware to hot-cold
D	Daily storage usage	1.7 PB	1.3 PB	Change eviction policy from LRU to least-rare
E	Daily storage usage	11.7 PB	10.7 PB	Change eviction policy from LRU to least-rare

Table 5: **Savings from continuous offline analysis.** 5 production buckets were optimized in a 1 month period baed on offline analysis results. The table shows the key metric being optimized and the value of the metric before and after optimization.

seeder functionality.

FaaSNet [17], also from Alibaba, takes an even more decentralized approach for distributing serverless containers, foregoing the use of all centralized nodes and instead utilizing a tree-based network overlay. Dadi [13] and VMThunder [19] also use a tree overlay to distribute container/VM images among peers, with cache misses serviced by nodes higher in the tree. These approaches are similar to Meta's StaticTree, except that StaticTree constructs locality-aware trees that minimize the network distance between sibling nodes.

Classically, many tree- and mesh-based networks have been proposed for high-bandwidth data distribution to many hosts [4,6,11,12]. While Owl uses a forest of distribution trees, these trees are per-data-chunk and ephemeral, with edges persisting only for the time needed for a peer to download a single chunk of data. The trees in prior works are longer-lasting and used for more than just a single data chunk.

In contrast to all of these prior distribution systems, Owl's control plane is significantly more centralized. Owl trackers make explicit decisions on behalf of peers about what data to cache and evict, where to download each chunk from, and how to retry failed downloads. Owl trackers consequently have a much more complete view of download and peer state, which allows for making more optimal, global decisions. Centralization also improves ease-of-management. The classic argument against centralizing the distribution control plane has been a projected scalability bottleneck; Owl refutes this argument by demonstrating that careful design can scale even a highly-centralized control plane to support millions of clients and hundreds of petabytes of data distributed per day. Additionally, Owl demonstrates more flexibility than prior systems through its customized policies; container provisioning is currently just a small portion of Owl's total workload.

The control plane and data plane taxonomy used in this paper comes from software defined networking (SDN). Recently, Google's Orion [8] demonstrated the benefits of centralizing the SDN control plane. Distribution and SDNs both provide routing and store-and-forward-style caching. Yet, Owl tracks each individual chunk of data at a level of detail that is infeasible for network packets. This is possible because Owl operates on much larger units of data.

5 Conclusion

Owl distributes over 800 PB of hot content per day to millions of peers at Meta. Owl combines a decentralized peer-to-peer data plane with a highly-centralized control plane in which trackers make detailed decisions for peers such as for where to fetch each chunk of data, how to retry failed fetches, and which chunks to cache in peer memory and storage. Owl is highly-customizable through tracker policies that allow a unique configuration for each type of client.

6 Acknowledgements

We thank our shepherd, Doug Terry, and the anonymous reviewers for providing valuable feedback that allowed us to improve this paper. We also thank CQ Tang for reading early iterations of the paper and helping us focus our discussion.

References

[1] Introducing Kraken, an open source peer-to-peer docker registry. https://eng.uber.com/introducing-kraken/.

[2] What is Dragonfly? https://d7y.io/en-us/docs/overview/what_is_dragonfly.html.

[3] Berg, B., Berger, D. S., McAllister, S., Grosof, I., Gunasekar, S., Lu, J., Uhlar, M., Carrid, J., Beckmann, N., Harchol-Balter, M., and Ganger, G. R. The CacheLib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (November 2020).

[4] Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., and Singh, A. Splitstream: High-bandwidth multicast in cooperative environments. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)* (October 2003).

[5] Chen, Z., Zhao, Y., Miao, X., Chen, Y., and Wang, Q. Rapid provisioning of cloud infrastructure leveraging peer-to-peer networks. In *Proceedings of the*

9th IEEE Conference on Distributed Computing Workshops (2009).

[6] CHU, Y.-H., RAO, S. G., SESHAN, S., AND ZHANG, H. Enabling conferencing applications on the internetusing an overlay multicast architecture. In *Proceedings of ACM SIGCOMM* (August 2001).

[7] COHEN, B. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems* (2003).

[8] FERGUSON, A. D., GRIBBLE, S., HONG, C.-Y., KILLIAN, C., MOHSIN, W., MUEHE, H., ONG, J., POUTIEVSKI, L., SINGH, A., VICISANO, L., ALIMI, R., CHEN, S. S., CONLEY, M., MANDAL, S., NAGARAJ, K., BOLLINENI, K. N., SABAA, A., ZHANG, S., ZHU, M., AND VAHDAT, A. Orion: Google's software-defined networking control plane. In *Proceesings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (April 2021).

[9] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference* (2010).

[10] JACOBSON, S. H. Analyzing the performance of local search algorithms using generalized hill climbing algorithms. *Essays and Surveys in Metaheuristics 14* (2002), 441–467.

[11] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND O'TOOLE, J. W. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (October 2000).

[12] KOSTIC, D., RODRIGUEZ, A., ALBRECHT, J., AND VAHDAT, A. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)* (October 2003).

[13] LI, H., YUAN, Y., DU, R., MA, K., LIU, L., AND HSU, W. DADI: Block-level image service for agile and elastic application deployment. In *Proceedings of the 2020 USENIX Annual Technical Conference* (July 2020).

[14] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the International Workshop on Peer-to-Peer Systems* (2002), pp. 53–65.

[15] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Monterey, CA, Oct. 2015).

[16] TANG, C., YU, K., VEERARAGHAVAN, K., KALDOR, J., MICHELSON, S., KOOBURAT, T., ANBUDURAI, A., CLARK, M., GOGIA, K., CHENG, L., CHRISTENSEN, B., GARTRELL, A., KHUTORNENKO, M., KULKARNI, S., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., VENKATESAN, V., AND ZHANG, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2020* (November 2020).

[17] WANG, A., CHANG, S., TIAN, H., WANG, H., YANG, H., LI, H., DU, R., AND CHENG, Y. FaasNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In *Proceedings of the 2021 USENIX Annual Technical Conference* (July 2021).

[18] YANG, J., YUE, Y., AND RAHMI, K. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (November 2020).

[19] ZHANG, Z. Z., LI, Z., WU, K., LI, D., LI, H., PENG, Y., AND LU, X. VMThunder: Fast provisioning of large-scale virtual machine clusters. *IEEE Transactions on Parallel and Distributed Systems 25*, 12 (2014), 3328––3338.

BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms

Benjamin Reidys* Jinghan Sun* Anirudh Badam† Shadi Noghabi† Jian Huang

University of Illinois at Urbana-Champaign †*Microsoft Research*

Abstract

Cloud platforms today make efficient use of storage resources by slicing them among multi-tenant applications on demand. However, our study discloses that cloud storage is still seriously underutilized for both allocated and unallocated storage. Although cloud providers have developed harvesting techniques to allow evictable virtual machines (VMs) to use unallocated resources, these techniques cannot be directly applied to storage resources, due to the lack of systematic support for the isolation of space, bandwidth, and data security in storage devices.

In this paper, we present BlockFlex, a learning-based storage harvesting framework, which can harvest available flash-based storage resources at a fine-grained granularity in modern cloud platforms. We rethink the abstractions of storage virtualization and enable transparent harvesting of both allocated and unallocated storage for evictable VMs. BlockFlex explores both heuristics and learning-based approaches to maximize the storage utilization, while ensuring the performance and security isolation between regular and evictable VMs at the storage device level. We develop BlockFlex with programmable solid-state drives (SSDs) and demonstrate its efficiency with various datacenter workloads.

1 Introduction

In modern cloud platforms, storage devices such as flash-based solid-state drives (SSDs) have been virtualized as system-wide shared resources to provide storage services across multiple application instances [5, 9, 14, 29, 38, 65]. This enables cloud platforms to make efficient use of storage capacity and bandwidth by slicing them among multiple multi-tenant virtual machines (VMs) [43, 60, 75]. However, our study of the event traces collected from popular cloud platforms [3, 9, 22] reveals that storage I/O is still significantly underutilized for both unallocated (unsold) and allocated storage. For instance, we find that 40% of the cloud storage servers have 25% of their storage unallocated, and the I/O utilization of allocated storage is under 33% on average (see Figure 1 and §2.1).

To improve the resource efficiency in the cloud, providers offer evictable VMs (i.e., Spot VMs or Harvest VMs) [4, 23, 62]. These evictable VMs allow users to use unallocated resources with low priority, i.e., the resources of evictable VMs can be reclaimed by regular VMs at any time. Recent studies [5, 48, 69, 76] advanced this technique by improving the resource allocation and scheduling for evictable VMs with heuristic-based harvesting approaches.

However, prior work on resource harvesting mainly focused on CPU and memory resources, which cannot be directly be applied to cloud storage for three reasons. First, current cloud storage virtualization approaches do not support storage harvesting, and dynamic reallocation of resources is not feasible. Second, cloud storage usually stores sensitive application data, which requires careful management for storage allocation and deallocation. Third, cloud storage can suffer from significant harvesting overhead due to the block erasure and metadata updates, which requires specific optimizations for enabling efficient storage harvesting.

In this paper, we present BlockFlex, the first learning-based storage harvesting framework, which enables transparent storage harvesting for both allocated and unallocated storage at a fine-grained granularity, while ensuring data privacy for cloud users with low harvesting overhead.

To develop BlockFlex, we first conduct a characterization study of storage resources that could be harvested in a cloud platform. According to our study (see §2), we find that for unallocated VMs configured with 512GB SSD, 78%, 43%, and 25% of them can be harvested and used for 1 hour, 6 hours, and 12 hours, respectively. This provides us the heuristic information about how these unallocated storage resources can be utilized. As for the allocated storage for VMs, an average of 70% can be harvested, however, the time available for harvesting varies depending on the workloads running in the VMs. Our study discloses the dynamics of available storage resources, which drives us to develop a learning-based approach for assisting the storage harvesting.

*Co-primary authors.

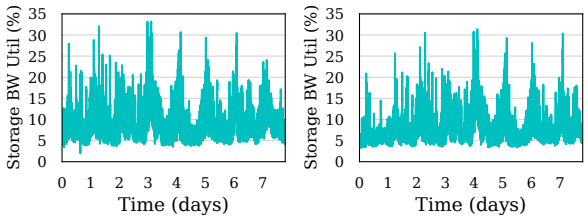
To enable transparent and fine-grained storage harvesting, we rethink the abstractions of storage virtualization for flash-based SSDs. The recent development of software-defined flash (SDF) in datacenters [29, 51] allows VMs to map their storage to dedicated flash channels. We build on top of the SDF abstraction and propose a new class of virtualized SSDs, named *ghost vSSD*. A ghost vSSD is created by harvesting free flash blocks from either unallocated or allocated but unused storage. The ghost vSSD provides the flexibility for fine-grained storage allocation and deallocation as well as block-level state tracking. It enables storage harvesting at the device level, which is transparent to the upper-level applications running on the VMs. Each ghost vSSD aims to meet the storage capacity and bandwidth requests from an evictable VM, however if needed, they can be reclaimed by regular VMs at any time.

However, frequent preemption and harvesting will inevitably introduce performance overheads to both regular VMs and evictable VMs, and even cause VM recreations. Therefore, it is desirable to provision the best-fit storage resource for an evictable VM. To achieve this, we develop learning-based techniques to predict the storage demands as well as the storage resources available for harvesting, in terms of storage capacity, bandwidth, and the duration available for harvesting. With these predictions, for each ghost vSSD, BlockFlex ensures the harvested storage resource will maximally meet the requirements of evictable VMs, while minimizing the opportunity of being preempted unexpectedly by regular VMs.

BlockFlex uses the Long Short-Term Memory (LSTM) network for online predictions at runtime, because of its low overhead and ability to make time-series predictions. We improve the prediction accuracy by developing different LSTM models for different dimensions of storage properties. For the predictions of storage capacity, bandwidth, and the time available for harvesting, BlockFlex can reach at 94.1%, 95.3%, and 93.1% accuracy, respectively, with slight over-provisioning. Upon mispredictions, BlockFlex implements different exception handlers for different cases (see the details in Table 1). As mispredictions do not happen frequently, the performance impact of misprediction handling is negligible in BlockFlex.

To minimize the performance interference between the regular VM and evictable VM caused by the storage harvesting, we assign higher priority to I/O requests from regular VMs when sharing the same SSDs with evictable VMs. When the harvested storage needs to be reclaimed, its flash blocks will be erased first to ensure data security, and then returned back to the corresponding regular VMs. Overall, we make the following contributions in this paper.

- We conduct a characterization study of the storage efficiency in different cloud platforms, our observations motivate the desirable need for storage harvesting.
- We rethink the abstractions of storage virtualization in modern cloud platforms for enabling fine-grained storage harvesting with software-defined flash.



(a) Storage utilization per VM. (b) Storage utilization per server.

Figure 1: The bandwidth utilization of allocated cloud storage.

- We build a learning-based storage harvesting framework named BlockFlex that can harvest both unallocated and allocated storage resources.
- We develop lightweight predictors that can make efficient predictions for both storage demand and availability in terms of storage capacity, bandwidth, and the time available for harvesting.
- We implement BlockFlex with real programmable SSDs and show its efficiency with various datacenter workloads.

Our experiments show that BlockFlex can improve the overall storage utilization by up to 1.75× in cloud platforms. BlockFlex is lightweight, it incurs trivial additional overheads to cloud platforms. BlockFlex can improve the performance of evictable VMs running with batch-processing workloads by 1.68× on average, while having negligible negative impact on the performance of regular VMs. The codebase of BlockFlex is available at https://github.com/platformxlab/blockflex.

2 Characterization for Storage Harvesting

Although storage virtualization has been widely deployed in cloud platforms, we observe that storage devices are still significantly underutilized, in terms of both storage bandwidth and capacity. In this section, we first quantify the cloud storage utilization, and then we conduct a hypothetical analysis of the opportunities for storage harvesting.

2.1 Cloud Storage Utilization

The storage underutilization in cloud platforms is due to both the poor utilization of allocated storage resources and the large portion of unallocated resources, as we discuss below.

Allocated storage resources. We conduct the storage utilization study based on the open-source cloud traces from Alibaba [3] and Google [22]. These traces track the usage of allocated storage resources across both VMs and physical servers. Alibaba cloud traces contain the VM utilization logs of 4K servers over 8 days, and Google cloud traces were collected from 12.5K servers over 29 days. As different cloud traces emphasize different aspects of the cloud storage usage (e.g., storage capacity, I/O bandwidth, server utilization, and

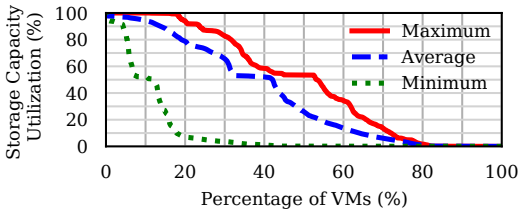


Figure 2: The capacity utilization of allocated cloud storage.

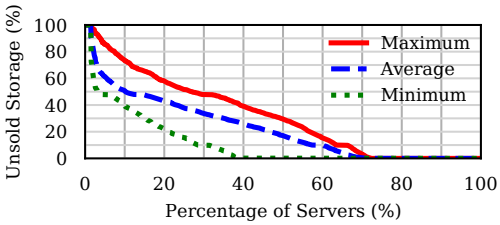


Figure 3: Unallocated storage in cloud servers.

VM utilization), we analyze both traces. We summarize our study results as follows:

- *Storage bandwidth:* We show the bandwidth utilization of Alibaba cloud [3] in Figure 1. The bandwidth utilization of allocated storage across all VMs is below 33%, and the average bandwidth utilization across all VMs over their entire lifetime is 9.2%. For physical servers that usually host multiple VMs, we obtain a similar trend: the bandwidth utilization of the physical storage devices is below 31%, and the average bandwidth utilization is 8.6%.
- *Storage capacity:* We present the cumulative distribution of storage capacity across the VMs of Google cloud [22] in Figure 2. We find that 20% of the VMs almost did not use their allocated storage capacity, 50% of the VMs used only 26.4% of the allocated storage capacity on average, and only 20% of the VMs used up to 90% of their allocated storage. Although different VMs may allocate different storage capacities, our study shows that their capacity utilization is surprisingly low.

The low utilization of allocated cloud storage resources is mainly due to two major reasons. First, cloud platforms usually allocate storage resource associated with each VM at a coarse-grained granularity for simplified storage management. For instance, the storage capacity of a VM in the Azure Cloud is linearly proportional to the number of allocated processor cores [5, 76], no matter whether the VM is I/O-intensive or CPU-intensive. Second, storage allocation is usually conducted in a static manner, while the storage usage of the workloads running in each VM changes dynamically over time. Therefore, the user of a VM has to over-provision sufficient storage for the peak demand upon VM creation.

Unallocated (unsold) storage resource. Beyond the allocated storage, the unallocated (unsold) storage in cloud platforms is another source for storage underutilization. This is because cloud providers usually over-provision VMs in their resource pool to satisfy the elasticity requirement from customers [5]. As each unsold VM consumes a fixed amount of resources

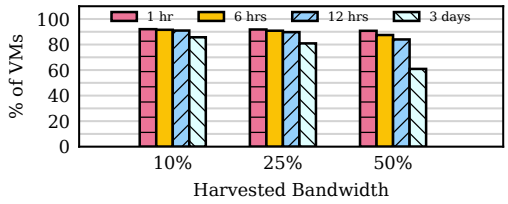


Figure 4: The availability of allocated storage for harvesting.

(e.g., processor cores, memory, and storage), it will result in storage resources unallocated.

To further understand the unallocated storage, we analyze the cloud traces of unsold storage resources from Azure Cloud [5]. The traces include the VM allocation/deallocation logs for about 1,400 servers over 24 hours. As shown in Figure 3, nearly 70% of cloud servers have unsold storage resources, 50% of the servers have an average of 17.3% of their storage unallocated, and 20% of the servers have at least 20.1% of their storage unallocated. Given that a datacenter has thousands of servers, the unallocated storage is another critical source for the storage underutilization.

2.2 Opportunities for Storage Harvesting

As discussed in §2.1, we identify two sources for storage harvesting: unallocated storage and allocated storage. In this part, we conduct a hypothetical analysis of these storage resources to understand their potential for storage harvesting.

Analysis methodology. We study the cloud traces as discussed in §2.1, with a focus on the storage resource allocation and deallocation. We analyze the available storage in allocated and unallocated VMs over time, and check (1) whether we can harvest storage from them for a hypothetical harvest VM requesting a certain amount of storage capacity; (2) how long the harvested storage can last; (3) how many storage resources we can potentially harvest for the hypothetical harvest VMs. Note that Google and Alibaba cloud traces only report normalized numbers, so we use percentages rather than absolute numbers in our analysis.

Allocated storage resource. We first apply the hypothetical analysis on the allocated storage resource. Given a hypothetical harvest VM requesting different percentages (10%, 25%, and 50%) of storage bandwidth from a regular VM, we investigate how many servers have such available bandwidth, and how long these resources are available for harvesting. We report the average percentage across the entire trace. The results are summarized in Figure 4. We observe that more than 91% of the servers have harvestable bandwidth for 12 hours, and about 76% of the servers have harvestable bandwidth for 3 days. As we harvest storage for a shorter time (i.e., less than 12 hours), the portion of the available servers is consistently high. This is due to the constant low storage utilization of allocated VMs, as shown in Figure 1.

Unallocated storage resource. We now explore the unallocated storage resource. Given a hypothetical harvest VM that

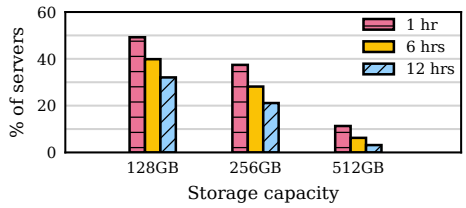


Figure 5: The availability of unallocated storage for harvesting.

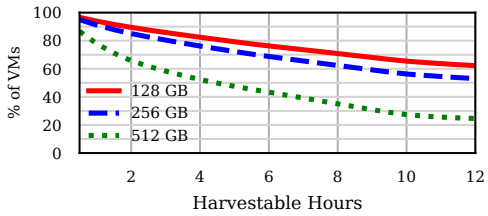


Figure 6: The availability of unsold regular VMs for storage harvesting with different capacities.

requests different storage capacities (128GB, 256GB, and 512GB), we analyze how many servers can satisfy the request from this harvest VM, and how long the available storage can last. We present the study results in Figure 5. Our study finds that 32% of the servers can satisfy the requirement of 128GB storage capacity for 12 hours. If the harvest VM requests storage for a shorter time, such as 1 hour, 50% of the servers can meet the request. As harvest VM increases the requested storage capacity, the number of harvestable servers decreases.

We also study unsold regular VMs. We vary the storage capacity request from 128GB to 512GB for the hypothetical harvest VM, and demonstrate our study results in Figure 6. For a hypothetical harvest VM of 128GB storage capacity, 94%, 76%, and 62% of the unsold regular VMs can be harvested for 1 hours, 6 hours, and 12 hours, respectively. As we increase the requested storage capacity for the harvest VM, the percentage of available unsold regular VMs drops. However, we still find a decent amount of unsold regular VMs can be harvested. For instance, for the harvest VM that requests 512GB storage capacity, 43% and 24% of the unsold VMs are available for 6 hours and 12 hours, respectively.

It is worth noting that the storage bandwidth is usually allocated proportionally with storage capacity in cloud platforms [19, 29]. This is also reflected in the cloud traces we studied in this paper. For instance, as for the VM with 128GB, 256GB, and 512GB, the storage bandwidth is 192 MB/s, 384 MB/s, and 768 MB/s, respectively. Thus, our study on the unsold storage capacity also applies to the storage bandwidth.

Takeaways. Our characterization study shows that:

- Both unallocated and allocated storage have sufficient storage capacity and bandwidth for harvesting, and they are available long enough to facilitate harvesting.
- The harvestable storage resource varies depending on the storage capacity and time available for harvesting. Harvesting a large storage capacity for a longer time has a lower chance of identifying the available storage resource.
- The harvestable storage resource from unallocated VMs and allocated VMs shows different availability patterns and trade-offs. We have a larger chance to harvest storage in allocated VMs, but this may have interference with the regular VM. The harvestable storage from unallocated VMs is limited, but it has no impact on the performance of regular VMs.

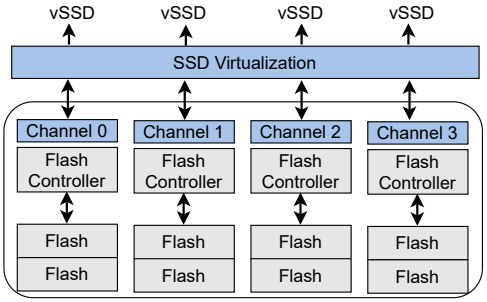


Figure 7: Storage virtualization with software-defined flash.

With BlockFlex, we aim to improve the cloud storage utilization by harvesting the available storage resources from both allocated and unallocated storage.

3 Technical Background

To facilitate our discussion, we first present the essential technical background of storage virtualization in cloud platforms, and then discuss the harvest VMs that will benefit from storage harvesting.

3.1 Storage Virtualization and SDF

In modern cloud platforms, storage virtualization has become the backbone of the storage infrastructures, in which storage devices such as flash-based solid-state drives (SSDs) are virtualized and shared by multiple VMs in order to improve storage utilization [29, 38, 60, 65]. The storage virtualization layer provides the system abstraction of virtualized storage devices (e.g., virtual disks) and hides the underlying hardware complexities from upper-level VMs. Each VM can have one or more virtual storage devices, and each virtual storage device can be mapped to one or more physical storage devices.

At the same time, SSDs are increasingly being adopted by cloud providers for their low latency and high throughput [1, 29, 30, 46]. Internally, an SSD consists of multiple flash channels, each channel has multiple flash chips, and each chip has thousands of flash blocks (see Figure 7). Each channel can issue I/O requests independently, thus, offering high parallelism and performance isolation. SSDs can only write data to free blocks, and once a free block is written, it is no longer available for future writes until it is erased. However, the erase operation is time-consuming. Thus, writes are issued to

flash blocks that have been erased in advance (i.e., out-of-place update). Because of this, SSDs employ a flash translation layer (FTL) to maintain the logical-to-physical address mapping, and manage the garbage collection (GC) operations.

To ultimately exploit the performance benefits of SSDs in the cloud, software-defined flash (SDF) was developed [39, 51]. In the context of SSD virtualization, SDF allows the upper-level VM to map its virtual SSD (vSSD) to a set of flash channels, as shown in Figure 7. Therefore, cloud providers can allocate storage capacity and bandwidth to each vSSD per its request by allocating fewer/more flash channels, following the pay-as-you-go model, while enabling the device-level performance isolation between vSSDs. The vSSD performs like a conventional storage disk, it provides the block interface to upper-level software, and uses a mapping table to index the logical-to-physical block address mappings [29, 51]. As SDF enables various cloud services such as Database-as-a-Service (DaaS) and Infrastructure-as-a-Service (IaaS) to achieve predictable storage performance and satisfy their service level objectives (SLOs), it has become an essential component in modern cloud platforms [16, 31, 55, 56, 64]. In this work, we develop BlockFlex based on the software-defined flash infrastructure.

3.2 Harvest Virtual Machine

To improve the resource utilization in cloud platforms, a few VM techniques have been developed recently [4, 5, 12, 20, 58, 62, 69]. Cloud providers offer *evictable VMs* or *Spot VMs* that run with lower priority than regular VMs, they can be evicted if resources are needed by a regular VM [4, 62]. With evictable VMs, cloud providers can sell unsold resources at a lower price while providing resource guarantees for regular VMs. Therefore, cloud customers usually rent evictable VMs to run batch-processing workloads or similar applications that have lower requirements on resource guarantees. Based on the evictable VMs, researchers developed *harvest VM* [5], *elastic VM* [69], and *memory-harvesting VM* [20], which further improve the cloud resource utilization by enabling flexible and dynamic harvesting of unallocated resources. To simplify the discussion, we will use harvest VM to represent these aforementioned VMs for resource harvesting in the remainder of the paper.

A majority of these harvest VMs were developed to harvest CPU and memory resources, and none of them can be directly applied to the storage resources. Additionally, prior work proposed various VM scheduling techniques by co-locating multi-tenant applications on the shared bare-metal servers to improve the resource efficiency [41, 43, 66, 71]. However, our study of various cloud traces discloses that the storage utilization is still a severe issue within modern cloud platforms. Since storage virtualization today assumes exclusive ownership of storage resources for each VM, it inevitably causes storage underutilization. In this work, we enable the storage harvesting for harvest VMs to improve the cloud storage utilization.

4 Design and Implementation

In this section, we first discuss the design goals and challenges of BlockFlex. After that, we will present the overview of the system as well as the design and implementation details of each component.

4.1 Design Goals and Challenges

As we develop BlockFlex to enable efficient storage harvesting, we aim to achieve the following goals:

- The storage harvesting should satisfy the storage requirements from harvest VMs while minimizing unexpected preemptions by regular VMs.
- The storage harvesting should be transparent to the upper-level VM to minimize changes to the VM and applications, as well as facilitate its production deployment.
- The storage harvesting should have minimal negative impact on the regular VMs to guarantee the quality of cloud services as we improve the global storage utilization.
- The storage harvesting should ensure the data safety, when it temporarily allocates unused data blocks from both allocated and unallocated storage to the harvest VMs.

Since cloud platforms today do not provide system support for storage harvesting, it is not easy to achieve the above goals. Additionally, existing resource harvesting techniques cannot be directly applied to storage resources. Specifically, we have to overcome the following challenges. First, cloud customers usually rely on the storage to permanently store their data, the data durability and availability are critical for storage services. This makes the storage harvesting fundamentally more challenging than the harvesting of CPU and memory resources. For example, shrinking available storage (upon reclamation) may result in data loss, while reclaiming memory and CPU resources mainly causes reduced performance. Second, the storage virtualization and management are different from that of CPU and memory resources, especially for SSDs that have intrinsic properties (see §3.1). Therefore, sharing storage resources while maintaining isolation among tenants needs new techniques. Third, storage allocation and deallocation usually incur more performance overhead than the context switch overhead caused by harvesting CPU and memory resources, which requires special development efforts for enabling the deployment of storage harvesting in cloud platforms.

4.2 System Overview

To the best of our knowledge, BlockFlex is the first storage harvesting framework built based on modern software-defined storage infrastructure. We present the system architecture of BlockFlex in Figure 8. To manage the harvested storage, we propose a new abstraction, named ghost vSSD (gSSD), on top of software-defined flash (§4.3). The ghost vSSDs

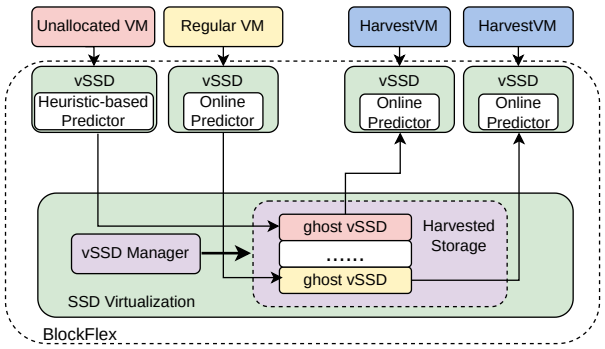
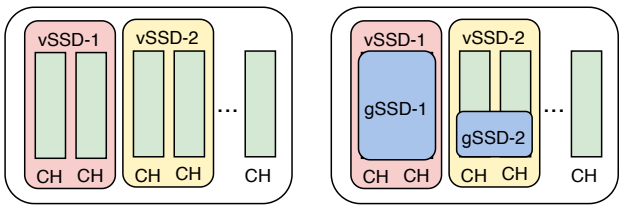


Figure 8: System overview of BlockFlex.



(a) Allocated and Unallocated vSSDs (b) Harvest available storage

Figure 9: Examples of harvesting storage. CH: flash channel; vSSD-1: unsold storage; vSSD-2: allocated storage; gSSD-1: harvest unsold storage; gSSD-2: harvest allocated storage.

can be attached to created vSSDs , therefore, no changes are required to VMs. BlockFlex will deploy a predictor in each vSSD (§4.4). For harvest VMs, BlockFlex will predict their demanded storage capacity and bandwidth, as well as how long the demand will last. For regular VMs, BlockFlex will predict their available storage capacity and bandwidth, as well as their available time. For unused storage resources, BlockFlex will use both heuristic-based approaches to predict the duration time available for harvesting. Based on the prediction, BlockFlex will make a best-fit match and allocate unused storage to the harvest VM. In case resource preemption happens to the harvest VM (caused by misprediction), BlockFlex will release the harvested storage to the regular VM and handle the exceptions for different scenarios (§4.5).

Since BlockFlex enables storage harvesting at the system virtualization level, it does not change the upper-level durability model (e.g., data replication) offered by current cloud storage infrastructures. For harvest VMs, cloud platforms assume their end users are aware of the relaxed durability guarantees and their applications may suffer from early reclamations. BlockFlex makes the best effort to allocate new gSSDs to ensure the data durability for harvest VMs. However, similar to Spot VMs [63], the owners of harvest VMs should be aware of the risk and take responsibility for their data as the cost of harvest VMs is much lower than regular VMs. As BlockFlex is deployed on top of existing software-defined storage infrastructure, it runs in a distributed environment where the global control plane manages the gSSDs and their allocations/deallocations. In the following section, we will discuss each technique proposed in BlockFlex, respectively.

4.3 New Abstraction for Storage Harvesting

As discussed in §3.1, with software-defined flash, the storage virtualization can map each virtual SSD to a number of flash channels depending on the storage capacity and bandwidth requested by the associated VM. We show two typical examples in Figure 9. Suppose we have a 1TB SSD that contains 16 channels. Each channel has 64GB and delivers a bandwidth of 70MB/s. As shown in Figure 9 (a), the cloud platform allocates two flash channels to vSSD-2 (128GB), leaving other flash channels temporarily unused (e.g., vSSD-1). Both vSSD-1 and vSSD-2 could provide opportunities for storage harvesting. For example, as shown in Figure 9(b), the entire unsold vSSD-1 (gSSD-1) and part of the allocated vSSD-2 (gSSD-2) could be harvested depending on their availability. The SDF offers the flexibility to allocate fewer/more resources to each gSSD.

However, the harvested storage still belongs to the original vSSDs, which could be preempted by existing or newly allocated regular VMs. Since the availability of harvested storage varies depending on the workloads in the cloud platform, it increases the complexity of storage harvesting.

4.3.1 Definition of Ghost vSSD

To simplify the management of harvested storage, we develop the gSSD abstraction. Its block interface is the same as that of the regular vSSD. Therefore, no code modifications are required for the VMs. Similar to vSSDs, each gSSD has a block-level mapping table to index the mappings of logical block addresses to physical block addresses , and a free block list to manage the free flash blocks. However, since each gSSD is created/borrowed from regular vSSDs and has a different lifetime (the time available for harvesting), we maintain a metadata structure for each gSSD, as shown in Figure 10.

The metadata of a gSSD includes its maximum bandwidth and capacity. We use the number of flash channels to represent the storage bandwidth, and the number of flash blocks to represent the storage capacity. As the actual storage bandwidth and capacity offered by a gSSD could vary at runtime, we use their maximum values because they are provided on a best-effort basis. We use the *expire* to indicate when the gSSD will no longer be available for use. This value is predicted with our duration predictor (see the detailed discussion in §4.4). The metadata structure also has a bit *in_use* to indicate whether the gSSD has been assigned to a harvest VM or not. If yes, the *vm_id* stores the ID of the corresponding harvest VM. The *home* pointer points to the regular vSSD from which the blocks in the gSSD are harvested. The *ghost* points to the created ghost vSSD after storage harvesting. The metadata is stored in the gSSD. It is initialized when the gSSD is created and updated when the gSSD is harvested/reclaimed.

```
typedef struct vmeta {
    int bandwidth          ; maximum bandwidth of gSSD
    int capacity           ; maximum capacity of gSSD
    int expire             ; how long the gSSD lasts
    boolean in_use         ; used by harvest VM or not
    string vm_id           ; harvest VM ID
    struct vssd* home      ; vSSD that owns these blocks
    struct vssd* ghost     ; points to the attached gSSD
} vmeta_t;
```

Figure 10: Metadata of a ghost vSSD in BlockFlex.

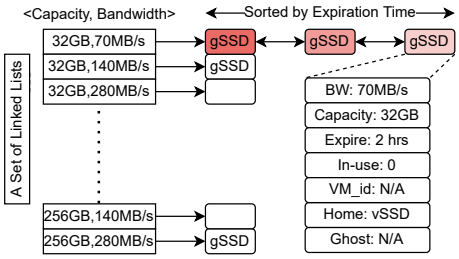


Figure 11: The organization of the gSSD pool in BlockFlex.

4.3.2 Management of Ghost vSSDs

We now discuss the gSSD creation and management.

Creating gSSDs. Instead of harvesting storage upon requests, BlockFlex allows regular vSSDs to proactively create gSSDs and add them into the gSSD pool managed by the vSSD manager (see Figure 8). This removes the harvesting procedure from the critical path. A vSSD creates a gSSD when its predictor predicts that it will have available storage resources for harvesting. These predictions occur at regular intervals (every three minutes by default). In order to create a new gSSD, BlockFlex will harvest free blocks from the vSSD and create a mapping table for them. Following our prior study on SDF [29], we use block-level address mapping tables for indexing flash blocks in the gSSDs/vSSDs. We align the address mapping granularity and flash erase granularity to simplify the storage management with improved efficiency. And each gSSD/vSSD has its own mapping table. Although the flash blocks of a gSSD could be harvested from a vSSD, the corresponding gSSD and vSSD will not share these harvested flash blocks. Therefore, we do not need to synchronize the mapping table entries between the gSSD and vSSD at runtime.

The metadata of a gSSD (Figure 10) is initialized with the number of flash channels harvested (*bandwidth*), the number of free blocks (*capacity*), and the predicted time the resources will be available for use (*expire*). The *home* of the gSSD will point to the regular vSSD, and the *ghost* will point to the newly created gSSD. At the same time, the gSSD will be added to the gSSD pool for serving future harvesting requests.

To simplify the management of gSSDs, we only create a gSSD when harvesting a chunk of resources. BlockFlex enables the storage harvesting at the granularity of a flash channel, 16GB size, and 30-minute for storage bandwidth, capacity, and duration time, respectively. To ensure reasonable performance isolation between regular VMs and harvest VMs, we restrict each vSSD to provide only one gSSD.

Managing gSSDs. To facilitate fast gSSD lookup, we organize gSSDs in a set of lists in the vSSD manager with considering the sorting in three dimensions: storage bandwidth, capacity, and time available for harvesting. We optimize the lists based on our observations that (1) the storage bandwidth and capacity are correlated with the number of channels available in a vSSD; (2) the time available for harvesting for each gSSD needs to be updated at regular intervals; and (3) we will not update the maximum storage capacity and bandwidth over the lifetime of a gSSD. Therefore, as shown in Figure 11, BlockFlex maintains a set of gSSD lists sorted by <capacity, bandwidth>. In each list, the gSSDs are sorted by their expiration time from the farthest one to the nearest one. There is a timer running periodically (per 15 minutes by default) to update the expire time in the gSSD pool. For the expired gSSDs but have not been allocated to any harvest VM, BlockFlex will remove them from the list.

Harvesting gSSDs. Upon receiving a request for storage harvesting, BlockFlex will check the gSSD pool to identify a best-fit match for the requested storage capacity, bandwidth, and time available for harvesting. BlockFlex uses the best-fit matching policy to minimize the waste of storage resources. These requested parameters are obtained from the predictors deployed in the vSSD of the corresponding harvest VM (see §4.4). Since the gSSD pool is sorted, we use the binary search to first locate the corresponding list that matches with the requested storage capacity and bandwidth. After that, we walk through the list until identifying an available gSSD whose expire time matches with the requested harvestable time.

Once a gSSD is identified in the pool, we set its *in_use* to 1 to indicate this gSSD has been assigned to a harvest VM and the corresponding harvest VM ID is recorded. BlockFlex supports concurrent gSSD allocations by managing the gSSD lists using non-blocking linked-lists implementation with the compare-and-swap operations [27]. Compared to the lifetime of a gSSD (hours or even days), the gSSD allocation overhead (a few microseconds) is trivial.

With a harvested gSSD, BlockFlex will assign its flash blocks to the vSSD of the corresponding harvest VM. This harvesting procedure is transparent to the harvest VM, as we track these blocks in the mapping table of the vSSD of the harvest VM, as shown in Figure 12. The address mapping table in the vSSD is extended to include the ID of the harvested gSSD. Therefore, upon data accesses from the harvest VM, its vSSD will conduct the address translation to translate the logical block address (LBA) to [gSSD-ID, gLBA]. With the obtained gSSD-ID, the corresponding gSSD will translate the gLBA to the physical block address (PBA). This enables BlockFlex to harvest multiple gSSDs for a harvest VM. With the expanded vSSD, the harvest VM can resize the vSSD and its file system with existing virtual disk and file system tools [13, 18, 67]. Note that the address mapping of a vSSD will also index the

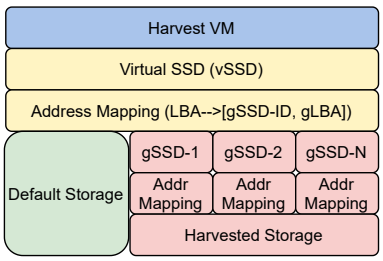


Figure 12: Harvesting multiple ghost vSSDs for a harvest VM.

default storage allocated when a harvest VM is created.

We assume each harvest VM will not request more than 256 gSSDs, so 1 byte is used to index the gSSDs. In total, each address mapping entry takes 9 bytes (4 bytes for LBA and 4 bytes for PBA). Given a harvest VM that requests 128GB storage, and each flash block is 4MB, the block-level address mapping of a vSSD will take only 288KB.

Reclaiming gSSDs. When a harvest VM finishes its jobs, the harvested gSSDs will be reclaimed to the pool in the vSSD manager. Upon the gSSD reclamation, the corresponding entries in the address mapping table of the vSSD will be removed. BlockFlex will check whether a gSSD will expire soon or not (i.e., in 30 minutes by default). If yes, BlockFlex will erase the flash blocks for data safety, and remove the gSSD instance. Otherwise, BlockFlex will add the gSSD into the gSSD pool for future harvesting. Since the erase operation is expensive, BlockFlex leverages the channel parallelism of an SSD to execute them in parallel.

The additional erase operations caused by gSSD reclamation has minimal impact on the lifetime of SSDs. This is for two major reasons. First, BlockFlex ensures wear leveling of SSDs by following a relaxed wear-leveling scheme proposed in our prior study [29]. It showed that SDF can achieve near-ideal SSD lifetime by swapping channels every 19 days on average for data center workloads, and 12 days on average for the worst case of erasing channels at full bandwidth. The wear leveling plays a fundamental role of ensuring the device lifetime, no matter whether flash blocks are used by regular VMs or harvest VMs. Second, the harvesting procedure itself only introduces erases when harvested storage is reclaimed, and it happens infrequently. Based on our study, for a given vSSD, it is harvested about every 2.1 days and consumes an average of 25% of the SSD (see §5.2), meaning the entire vSSD is erased once per 8.4 days. For modern SSDs that usually have 10K P/E cycles and can last 5-year lifetime, the storage harvesting operations will consume about 2% of the device lifetime, which is acceptable in practice.

In addition, with the assistance of predictors (see §4.4), BlockFlex minimizes the chances of early reclamation, and takes the erase operations from the critical path. However, a reclamation would still happen, even though a gSSD is in use by a harvest VM. This could be caused by the resource preemption issued by a regular VM. We will discuss how BlockFlex handles this in details in §4.5.

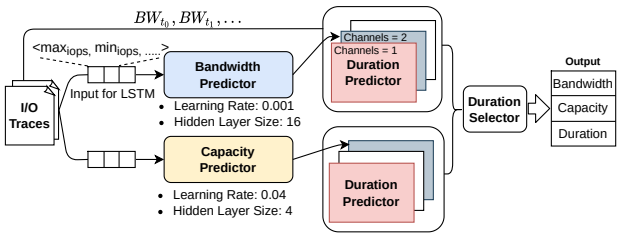


Figure 13: The workflow of the predictors used in BlockFlex.

4.4 Predictions for Storage Harvesting

Instead of relying on the cloud customers or VM users to specify their demanded or unused storage resource, we use a lightweight online learning approach to predict them. As discussed in §4.2, each vSSD has an online predictor, except those for the unallocated (unsold) VMs.

4.4.1 Heuristic-based Prediction for Unsold VMs

For the unallocated (unsold) VMs, we use a heuristic-based approach, based on our study characterizing the unallocated storage in cloud platforms (see §2). Recall that cloud providers usually over-provision VMs to provide the elasticity for their services. They reserve different regular VMs with various storage capacities. The common sizes include 128GB, 256GB, and 512GB for simplified VM management and deployment. According to our study in Figure 6, their availability for harvesting varies by their capacities.

Previous harvesting studies have identified that past values are a useful indicator for the available time of unsold storage [5]. In our study of unsold storage resource, we confirm that the available time of unsold storage for harvesting is stable. For this reason, we tag each unsold VM with a predicted duration time using the histogram of previous available times for the unsold VM with the same storage capacity. For instance, for the unsold VMs with 512GB storage capacity, we can use 20% of them as gSSDs that would be available for 12 hours, 20% for 6 hours, and the remaining for 1 hour. This distribution could change depending on the heuristic study of the corresponding cloud platform. The distribution of these gSSD sizes depends on the configured storage capacities for the unsold VMs.

4.4.2 Online Learning for Allocated and Harvest VMs

We predict the harvestable storage resource for allocated VMs, and demanded storage resource for harvest VMs. Since the predictions for allocated VMs and harvest VMs are both determined by their workloads, they use the same learning-based approach but different learning parameters.

We show the entire prediction workflow of BlockFlex in Figure 13. In each vSSD, we collect the read, write, and erase operations at the block layer for online predictions, therefore, we do not rely on the systems software running on top of

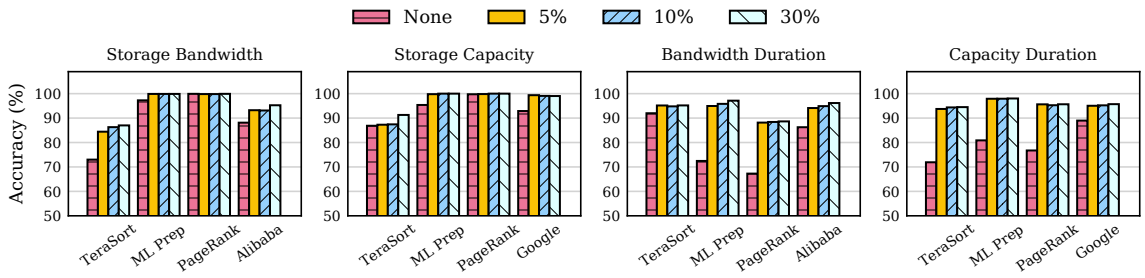


Figure 14: Prediction accuracy of storage bandwidth, capacity, and duration time available for harvesting, with various over-provisioning ratios. A slight over-provisioning for storage harvesting can significantly improve the prediction accuracy.

the vSSD. Based on these I/O traces, we infer the bandwidth, throughput (IOPS), and current storage utilization.

We use Long-Short Term Memory (LSTM) models [28] to develop our predictors, because of their strength in time-series predictions and relatively low overhead. The inputs for LSTMs are statistical measures gathered from the bandwidth, IOPS (e.g., max_{iops}, min_{iops}), and storage utilization. By default, BlockFlex trains the models every three minutes using the collected statistics from the preceding 15 minutes. This introduces minimal performance and memory overhead. Both bandwidth predictor and capacity predictor use the same LSTM model, but we tune their learning rate and hidden layer size slightly differently for improved accuracy (see Figure 13). These predictors will generate the predicted bandwidth (in channels) and predicted capacity (in GB), respectively.

The predictions of storage bandwidth and capacity are passed to their respective duration predictors. Each duration predictor consists of a collection of individual sub-predictors. Each sub-predictor is responsible for a possible output from the bandwidth/capacity predictors. For instance, if the output of the bandwidth predictor ranges from 1 to 16 channels, we will have 16 duration sub-predictors, each sub-predictor will predict its corresponding duration time by using the history of previous durations at that demand.

To ensure a gSSD can satisfy both the storage and bandwidth requirements from a harvest VM, the duration selector takes the maximum duration for demanded storage resources. To ensure a regular VM will not reclaim resources early, the selector takes the minimum duration for the harvestable storage resources. The final output delivered by the predictors in BlockFlex is presented in a tuple of <bandwidth, capacity, duration>. We describe the details of each predictor as follows.

Storage bandwidth: For the prediction of storage bandwidth, we use six inputs for the LSTM model: the *maximum, minimum, and average* for both bandwidth and IOPS. We do not use other statistical measures as inputs because they do not improve the prediction accuracy and slow down the convergence of the model. As the number of flash channels is proportional to the storage bandwidth, we use the number of channels as the bandwidth metric to simplify the bandwidth prediction.

Storage capacity: The prediction model for the storage capacity is similar to the model used for the storage bandwidth. We use the *maximum, minimum, and average* of past storage utilizations, and the *current changes in storage utilization* as the inputs. We find that using the changes in storage utilization helps differentiate long periods of sequential writes against shorter changes. We use the number of flash blocks as the output of the capacity predictor.

Duration: For the duration, we make the predictions for storage bandwidth and capacity separately. For allocated VMs, we predict how long their available storage capacity and bandwidth can be used by harvest VMs; for harvest VMs, we predict how long a demand of storage capacity and bandwidth will last before more resources are needed. As discussed, a set of sub-predictors are used for each demanded bandwidth/capacity. BlockFlex updates and maintains the history of durations for model training and inference.

4.4.3 Resource Provisioning for Improved Accuracy

We examine the accuracy of the LSTM models we develop for the aforementioned predictors using various cloud workloads (see their descriptions in Table 2). A prediction for storage bandwidth and capacity is considered accurate if the predictor predicts at least as much as the actually demanded/available storage. A prediction of duration time is considered accurate if the predicted storage bandwidth and capacity lasts as long as the actual demand/availability. We track the actual storage demand/availability and predicted storage demand/availability to calculate the prediction accuracies.

As shown in Figure 14, the average accuracies of predicting storage bandwidth, capacity, and their durations are 89%, 93%, 79%, and 79% on average. Their accuracy varies for different workloads. To further improve the prediction accuracy and avoid resource preemptions (see §4.5), we use a simple yet effective approach – over-provisioning more storage resources based on the predictions of demanded storage resources, and under-provisioning storage resources based on the predictions of harvestable storage resources. We vary the provisioning ratio from 5% to 30%, and show the updated accuracies in Figure 14. We find the accuracies of all the predictions can reach 93–96% with a provisioning ratio of 5%. As we increase the provisioning ratio, we do not see much accuracy improvement. Therefore, we use the 5% provisioning ratio in BlockFlex by default.

Table 1: Exception handling for different scenarios.

ID	Harvestable Storage	Demanded Storage	Possible Exceptions
❶	Over-predict	Over-predict	Waste or Early Reclamation or N/A
❷	Over-predict	Under-predict	Under-Harvest or Early Reclamation
❸	Under-predict	Over-predict	Waste
❹	Under-predict	Under-predict	Under-Harvest or Waste or N/A

4.5 Exception Handling in Storage Harvesting

Although the predictors in BlockFlex deliver high accuracy as discussed in §4.4, mispredictions can still happen, causing exceptions during storage harvesting. Typical exceptions include the resource preemption in which a regular VM prematurely reclaims the harvested storage from a harvest VM, and under-harvesting in which a harvest VM must request additional storage resources to satisfy the request of more storage resource than the predicted demand. VM terminations and data loss could happen if these exceptions are not handled properly.

Misprediction types. Mispredictions can be categorized into two types: over-prediction and under-prediction. As we make predictions for both harvestable storage (in the regular VMs) and demanded storage (in the harvest VMs), the two misprediction categories apply to both sides, as shown in Table 1.

An over-prediction of demanded storage means that a harvest VM harvests more storage resources than it really needs; an under-prediction of demanded storage means that a harvest VM harvests less storage resources than it really needs. In contrast, an over-prediction of harvestable storage means that a regular VM has less harvestable storage resources than predicted; an under-prediction of harvestable storage means that a regular VM has more harvestable storage resources than predicted. During storage harvesting, any misprediction or combinations of mispredictions could cause an exception. BlockFlex employs different exception handling for each scenario.

Exception handling. As shown in Table 1[1], mispredictions could mainly cause three exceptions: *waste of storage resources*, *early resource reclamation*, and *under-harvesting*.

Waste of storage resources. BlockFlex could waste storage resources when mispredictions leave them unused. In the case ❶ of Table 1, a regular VM provides the storage resource requested from the harvest VM, although the harvest VM may over-predict its demanded storage resource. In case ❸, the waste of storage resources becomes worse, because the regular VM actually has more harvestable storage resources than the requested resources from the harvest VM. As we trade the over-provisioning of demanded storage in the harvest VMs for increased prediction accuracy, it is inevitable to cause some waste of storage resources. However, since BlockFlex uses a 5% over-provisioning ratio (see §4.4) in its predictors, the waste is minimal. Compared to the cloud platforms without storage harvesting, BlockFlex still improves the storage utilization. Therefore, BlockFlex does perform special exception handling for this exception.

Early resource reclamation. This could happen when a regular VM has less harvestable storage resources than the demanded storage resources from a harvest VM. Typical examples include the case ❶ and ❷ in Table 1, in which we over-predict the harvestable storage resource in a regular VM, but in reality, the regular VM has less harvestable storage than the demanded storage from a harvest VM. In both cases, the regular VM has to reclaim its storage from the harvest VM. To handle this exception, BlockFlex will identify a new gSSD that meets the requirements for storage capacity, bandwidth, and duration. After that, BlockFlex will copy all the data from the old gSSD to the new gSSD and update the address mapping table in the vSSD of the corresponding harvest VM. BlockFlex will migrate data between gSSDs at block granularity to minimize the impact on the running applications. BlockFlex will reclaim the old gSSD while ensuring its flash blocks are erased before being used by the regular vSSD (see §4.3.2). However, if there is no satisfactory gSSD available, an exception will be reported to the end users of the harvest VM (like what is done today for spot VMs [63]).

Under-harvesting. The exception of under-harvesting could happen when a harvest VM under-predicts its demanded storage resources (i.e., it requests less storage resources than it really needs). Typical examples include the cases ❷ and ❹. For the case ❷, under-harvesting could happen when the harvest VM under-predicts its demanded storage resources. For the case ❹, although the regular VM under-predicts its harvestable storage resources, the demanded storage in reality could still be more than the available storage resources in the regular VM. To handle this exception, BlockFlex will harvest new gSSDs for the harvest VM until meeting the demand. As discussed in Figure 12, BlockFlex enables the use of multiple gSSDs in a single vSSD. However, if there is no gSSD available, BlockFlex will report an exception to the users of the harvest VM, resulting in a termination of the harvest VM or a delay of job executions in the harvest VM.

Note that mispredictions could happen along all three dimensions (i.e., storage capacity, bandwidth, and time available for harvesting) of the storage resource. The described exception handling is used in BlockFlex for mispredictions along any of the three dimensions.

4.6 Implementation Details

We implement the gSSD abstraction of BlockFlex using a programmable SSD with 1TB capacity. The SSD has 16 channels, each channel has 4 dies, each die has 4 planes, each plane has 1024 blocks. Each block consists 256 pages, each 16KB. Its controller allows read/write/erase operations against the raw flash resources and enables the host to develop their own FTL for address translation, GC, and wear leveling.

The gSSD implementation takes 4.1K lines of code (LoC) using C programming language. The vSSD used in this paper

[1] If the demanded storage resource from a harvest VM exactly matches with the harvestable storage resource in a regular VM, there is no exception (N/A).

Table 2: Workloads used in our evaluation.

Workload	Description
TeraSort [25]	Sort data generated by TeraGen.
ML Prep [2]	Preprocess images for machine learning tasks.
PageRank [24]	Compute the pagerank of a graph.
YCSB [73]	Transaction processing on a database.

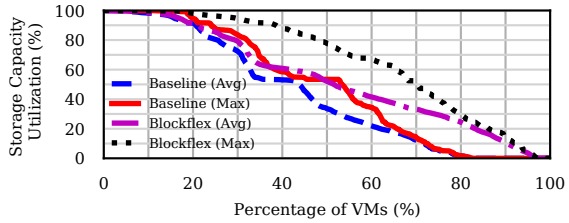


Figure 15: Improved utilization for underutilized storage.

is similar to the virtualized SSDs in our prior work [29]. BlockFlex creates different vSSDs for harvest VMs and regular VMs. It allocates physical flash channels for each vSSD to ensure performance isolation. Upon workload execution, BlockFlex handles the logical block I/O requests received by the vSSDs with actual read and write operations to the allocated physical flash blocks. We run BlockFlex on a real server with 8 Intel(R) Xeon(R) CPU E3-1240 v5 cores running at 3.5 GHz.

BlockFlex's predictors are implemented using PyTorch v1.9.0 [52] in 2.8K LoC using Python. Each model is implemented with one hidden LSTM layer fully connected with the input and output layers. The bandwidth and space predictors have an additional softmax layer applied to the output. All models use *adam* [36] as an optimizer and mean squared error as a loss function. We vary the learning rate and sizes of the hidden layer. Bandwidth prediction uses a learning rate of 0.005 and 16 hidden nodes. Capacity prediction uses a learning rate of 0.04 and 4 hidden nodes. Bandwidth duration uses a learning rate of 0.006 and 50 hidden nodes. Capacity duration uses a learning rate of 0.001 and 50 hidden nodes.

5 Evaluation

Our evaluation demonstrates that: (1) BlockFlex improves the storage utilization in cloud platforms by leveraging both underutilized and unallocated storage resources (§5.2); (2) BlockFlex improves the performance of harvest VMs while minimizing the impact on regular VMs (§5.3 and §5.4); (3) BlockFlex introduces negligible overhead to storage management (§5.5);

5.1 Experimental Setup

We evaluate BlockFlex with a set of synthetic workloads and real-world applications as shown in Table 2. We use Hadoop's TeraSort [25], ML Prep [2], and the PageRank implementation in GraphChi [24] to represent common applications in harvest VMs, while YCSB [73] represents common regular VM workloads. For TeraSort, we generate and sort 75GB datasets with

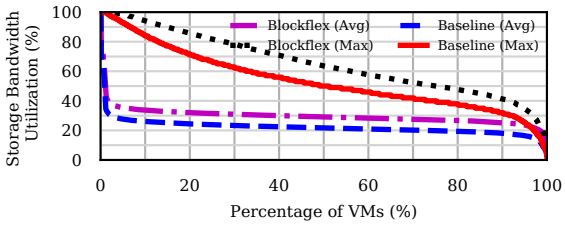


Figure 16: Improved utilization for underutilized bandwidth.

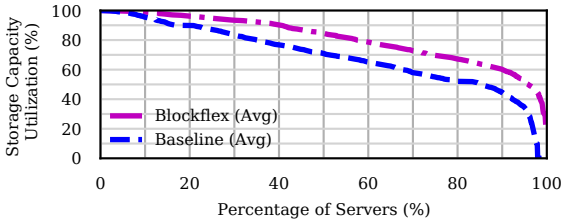


Figure 17: Improved utilization for unallocated resources.

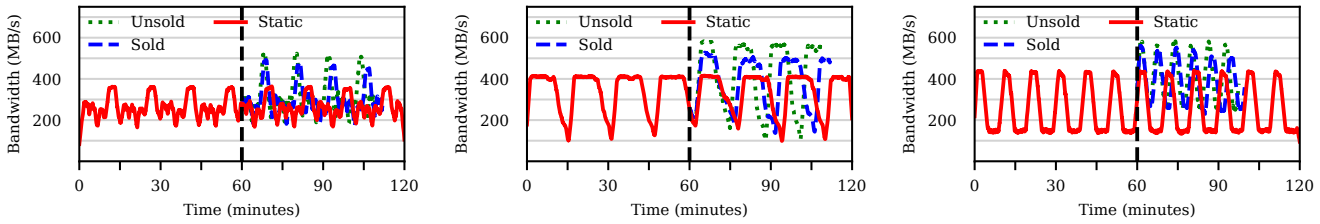
the TeraGen in Hadoop [25]. For PageRank, we use the Friendster graph (61GB) [72]. For ML Prep, we process images from the ImageNet data set (220 GB) [17]. For YCSB, we populate a key-value store RocksDB [54] with 180GB of data and run workloads A-E. In the evaluation, we report the numbers for YCSB-A since the workloads B-E deliver similar results.

5.2 Improved Storage Utilization

To evaluate the improved utilization of BlockFlex, we gather requests from 60,000 low priority VMs from Google traces to characterize the demand of harvest VMs. Their storage requests vary between 32GB and 512GB, and last between 30 minutes and 8.5 days (2.1 days on average). The demanded bandwidth is proportional to the demanded storage. We match these storage demands with harvestable storage capacity from 4,000 regular VMs. When evaluating the benefits of utilizing unallocated storage, we match them with unallocated VMs of 1,400 servers. Since VMs with low storage utilization present a greater opportunity for harvesting, we highlight the capability of utilizing the heavily underutilized storage with BlockFlex.

Underutilized Capacity. We first analyze the impact on the underutilized storage capacity, summarized in Figure 15. We compare the average and maximum utilization when using BlockFlex against the baseline utilization for VMs without harvesting (originally shown in Figure 2). We see an average improvement of 1.25× (43% vs. 54% utilization) across all VMs and an improvement of 1.75× (20% vs. 35%) for those that had less than 60% storage utilization. This shows the benefits BlockFlex can obtain, especially when harvesting flash blocks from VMs with low storage utilization.

Next, we see that the maximum utilization across all of the VMs is increased by 1.37× (49% vs. 67%). We also observe that the over-provisioning we add to the predictions ensures that we do not fully utilize any regular VM. This reinforces



(a) TeraSort (b) ML Prep (c) PageRank

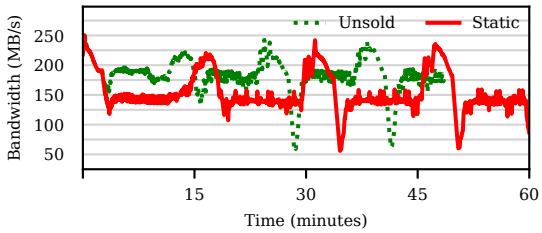
Figure 18: Performance benefits of storage harvesting for harvest VMs.

that BlockFlex has a low probability of reclamation.

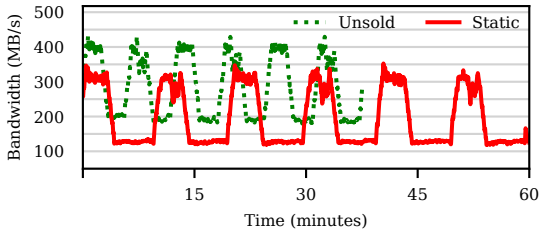
Underutilized Bandwidth. We now analyze the underutilized storage resource from a bandwidth perspective, summarized in Figure 16. Our results show a stable improvement of 1.34× (22% vs. 30%) for all VMs. BlockFlex also increases the maximum utilization by 1.27× (53% vs. 66%). As with underutilized storage, we avoid reclamations by not fully utilizing the bandwidth of regular VMs. This demonstrates that BlockFlex can improve both the bandwidth and capacity utilization of cloud storage from underutilized resources.

Unallocated Storage. We analyze the utilization improvement by harvesting unallocated VMs, presented in Figure 17. We observe that BlockFlex improves the overall utilization by 1.17× (69% vs. 81%). Servers with utilization below 60% are improved by 1.42× (45% vs. 64%).

For underutilized and unallocated storage resources, we observe 1.25× improvement on average, showing that BlockFlex can significantly use both underutilized and unsold storage resources to improve utilization. For extremely underutilized cases (under 60%), we observe 1.48× improvement on average. This shows that BlockFlex can successfully match the harvestable storage resources to the demands from harvest VMs.



(a) ML Prep Bandwidth



(b) PageRank Bandwidth

Figure 19: Read bandwidth of ML Prep and PageRank workloads after storage harvesting.

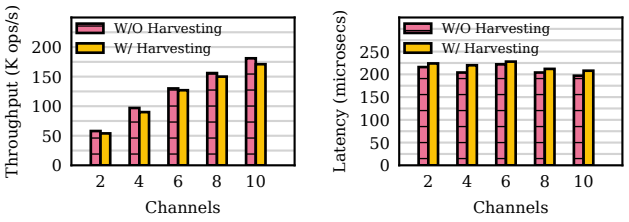
5.3 Improved Performance for Harvest VM

We examine how BlockFlex improves the performance of harvest VMs. The results are shown in Figure 18. We evaluate three different configurations: **Static:** the harvest VM is statically configured with 8 channels and does not harvest. This represents the current (baseline) storage virtualization. **Sold:** a 4-channel gSSD is allocated from channels occupied by a regular VM that uses 50% of its maximum bandwidth. **Unsold:** a 4-channel gSSD is allocated from unallocated channels. For both unsold and static, the gSSD is harvested after one hour. Before each experiment, we warm up the SSD to ensure GC will occur. We run all workloads for two hours.

By harvesting additional channels, the harvest VM has significantly improved bandwidth. As we compare the Sold scheme with the Static scheme, the workload performance is improved by 16–51% on average. For the Unsold scheme, the lack of interference with the regular VM improves the storage bandwidth by 22–60%. We observe the best improvement for PageRank, as its workload spends more time on I/O than TeraSort or ML Prep workloads. The Unsold scheme provides an additional 6% bandwidth improvement over the Sold scheme on average. As we translate this into the end-to-end execution time, we see an average performance improvement of 20% using Sold storage, and 25% improved performance using Unsold storage. This demonstrates the significant performance benefits BlockFlex can obtain for IO-intensive applications, when utilizing either sold or unsold storage.

Clearly, additional flash channels can benefit write-heavy workloads, as we increase the I/O parallelism. It is less clear whether additional channels can benefit read heavy workloads, as the harvested channels cannot immediately satisfy reads. To investigate this, we focus on the read bandwidth improvements in Figure 19. For both ML Prep and PageRank workloads, we see an increase of 10−21% after 5 minutes of harvesting. After the full 60 minutes, the average increase of the read bandwidth stabilizes and reaches an overall improvement of 22−60%.

Specifically, for ML Prep, we see a slight increase (10%) as



(a) Bandwidth (b) 95th Percentile Latency

Figure 20: Performance of a regular SSD with storage harvesting enabled.

we redirect writes to the additional channels immediately upon harvesting (0-5 minutes). Afterwards, as we start issuing writes and reads to the new channels, we see the read bandwidth benefit stabilizes at an improved level (24%). As for the PageRank workload, it shows relatively consistent benefit in the read bandwidth (60%). This is because PageRank workload is write intensive, during the first two minutes of harvesting. Thus, the PageRank data is aggressively written to the new harvested channels, which benefits the read bandwidth in return.

5.4 Performance Impact on Regular VM

To investigate the impact of storage harvesting on regular VMs, we examine the interference generated by the harvest VM. We run the YCSB workload-A with 10 threads in the regular VM, and vary the number of flash channels in its vSSD from 2 to 10. The database tables are striped across all the available channels in the vSSD. We first measure the throughput and tail latency (95th percentile latency) of the YCSB workload without enabling storage harvesting. After that, we create a harvest VM to run the ML Prep workload. The harvest VM will harvest all the channels of the regular VM, and we measure the performance of the regular VM after the harvesting.

As shown in Figure 20, the throughput of YCSB Workload-A decreases slightly, while the latency is almost constant as we increase the number of channels. The storage harvesting does not introduce much overhead (5.1% on average), since the regular VM always has the higher priority for its I/O requests and available storage bandwidth. We observe a similar overhead for the tail latency, demonstrating that the storage harvesting has negligible negative impact on the performance of regular VMs.

We also examine the interference caused by additional GC and storage reclamations. As indicated in §5.3, GC is enabled in our experiments. We believe the GC overhead can be further reduced with erase suspension available in modern SSDs [35, 70]. We wish to explore this feature in our future work. As for the overhead caused by storage reclamations, we observe that reclaiming an entire flash channel results in 1.5% slowdown in the average bandwidth of regular VMs. Such an overhead is acceptable in reality, as storage reclamations do not happen frequently over the entire lifetime of VMs.

Table 3: Learning overheads for each iteration in our predictors

Predictor	Training Time (millisecs)	Inference Time (millisecs)	Model Size (KB)
Bandwidth	10.3	2.5	22
Space	13.0	0.4	12
Bandwidth Duration	410.0	4.1	1153
Space Duration	42.0	0.3	510
Total	**475**	**7.3**	**1697**

5.5 Overhead Sources in BlockFlex

We now profile the overheads introduced by BlockFlex. We begin by analyzing the overheads introduced by the predictors. We present the summary of these overheads in Table 3. First, we measure the time consumed by training each predictor for one iteration of online training. As discussed in §4.4, each model is trained one iteration every three minutes. Since each duration predictor has multiple models, their training is more expensive than storage bandwidth and capacity predictors. In total, training all of the predictors consumes 0.48 seconds on our multi-core server. In this case, cloud platform operators do not need powerful hardware accelerators like GPUs to deploy BlockFlex. Since input sizes and training frequency do not change by workload, the training overhead is the same across all the workloads evaluated in this paper.

For each inference, the total execution time is 7.3 milliseconds. This overhead is also incurred once every three minutes, but can be further optimized. For example, we can decrease the inference frequency when a vSSD has generated a gSSD.

To store the predictors for each vSSD, BlockFlex allocates about 1.7MB memory space. It also allocates 4KB memory to store the history of bandwidth/capacity information used for training each iteration. This demonstrates the minimal performance and storage overheads of the predictors in BlockFlex.

We also profile the overheads of gSSD creation and lookup. They include the overheads of creating a new gSSD and harvesting free blocks from a regular vSSD. Since they only involve metadata operations, the overhead is 61 μs for creating a gSSD with 64GB. As gSSDs are created in the background, their creation overhead is not on the critical path. We organize the gSSD pool in sorted lists, the gSSD lookup takes 1.2 μs on average.

As we reclaim a gSSD from a harvest VM, its primary cost is on the erase of all the written blocks. Since we can parallelize the erase operations across channels, the limiting factor is the channel with the most allocated blocks. The total overhead is 17.1, 34.2, and 68.4 seconds for a channel (64GB) with 25%, 50%, 100% harvested, respectively. According to our study of various cloud traces, we observe that storage harvesting is infrequent (once every few hours). Additionally, compared to the lifetime of VMs in the cloud, the overhead of storage reclamation is relatively small, which has negligible impact on the performance of regular VMs.

6 Discussion and Future Work

Security implications of storage harvesting. A few potential security concerns may arise when sharing physical flash blocks in a cloud environment. First, we consider whether data could be leaked via harvested blocks. Since BlockFlex erases the flash blocks before creating/reclaiming the gSSD, it guarantees that user data will not be leaked through the storage harvesting. Second, we consider whether information could be leaked through the cached data, such as LBA-PBA mappings. As existing cloud infrastructure prevents access to the SSD virtualization, device driver, and controller layers without permission checking, therefore, even though a flash channel is shared across VMs, their accesses are protected. Third, we consider whether multiple VMs sharing a physical flash channel could suffer from side-channel attacks. It is actually hard for attackers to obtain meaningful information, since the variations could be caused by many factors, such as the number of co-located VMs or the CPU/memory contention.

Compatible with compute and memory harvesting. Upon the creation of harvest VMs, cloud platforms will allocate essential compute, memory, and storage resources. BlockFlex mainly targets storage harvesting to improve the overall cloud storage utilization, and improve the performance of applications bottlenecked by storage resources. It is compatible with prior studies on compute and memory harvesting [20, 69] for improving the whole-system resource utilization.

Semantic-aware storage harvesting. BlockFlex utilizes the vSSD interface in its implementation, making it transparent to applications in VMs. However, due to the lack of semantic information from upper-level applications, BlockFlex has to rely on the predictors to decide the harvestable and demanded storage resources. Additionally, preventing data loss is one of the key challenges when developing BlockFlex, allowing systems software to manage their data in harvested storage would be an alternative solution to address this challenge. Therefore, new APIs can be developed and exposed to popular software systems such as key-value stores and Hadoop Distributed File System (HDFS), which offers more flexibility for applications to manage their data in harvested storage.

7 Related Work

Storage virtualization and efficiency. Storage devices such as SSDs have been virtualized as system-wide shared resources for improved utilization in cloud platforms [29, 34, 43, 60, 61, 75]. Based on this, most recent studies focused on improving the performance isolation between collocated applications [6, 32, 33, 42, 49, 65]. However, our study (see §2) reveals that the cloud storage is still significantly underutilized. Ouyang et al. [51] identified the resource underutilization in the SSDs and developed the software-defined flash for cloud platforms. Similar to software-defined networking, software-defined flash is becoming a backbone technique in datacenters today [16, 56, 65]. However, most of them still use a static-allocation approach, which inevitably causes the waste of both storage capacity and bandwidth [11, 51]. Disaggregated storage architectures are proposed [40, 47, 50, 57, 68]. However, they still suffer from storage underutilization when we allocate disaggregated storage to VMs, due to the dynamic workload changes in VMs. BlockFlex addresses the storage underutilization problem by enabling storage harvesting in software-defined datacenters.

Resource harvesting in cloud platforms. Harvesting resources for VMs to improve the resource utilization is not a new concept in cloud platforms. Similar to the harvest VM, many studies have been developed recently, such as Spot VMs and burstable VMs [5, 7, 8, 20, 21, 59, 69]. However, they typically harvest compute and memory resources at a VM granularity. BlockFlex is the first work that focuses on storage harvesting, and addresses the unique challenges in storage harvesting and exception handling. Beyond harvesting unsold resources [5], we can also harvest underutilized allocated storage resources, while providing the performance and security isolation between regular VMs and harvest VMs.

Learning approaches for resource efficiency. Most recently, researchers started to leverage learning techniques to improve the task scheduling [53, 69, 77], cluster resource management [5, 10, 15, 45, 74], and performance optimizations [26, 37, 44, 78]. They showed that the learning-based approach is a promising method to address system optimization problems. However, it is still unclear how they can benefit the cloud storage. In this work, we apply the learning-based approach to improve the storage utilization within our storage harvesting framework. We customize the classical LSTM models for the predictions of harvestable and demanded storage resources, and show their efficiency in our evaluation.

8 Conclusion

In this paper, we first conduct a characterization study of the cloud storage utilization, and discloses that the low storage utilization exists pervasively in modern cloud platforms. To this end, we develop a learning-based storage harvesting framework BlockFlex, which can harvest both allocated and unallocated storage for evictable VMs. Our experiments show that BlockFlex can significantly improve the cloud storage utilization, while accelerating the storage performance of harvest VMs with minimal impact on the regular VMs.

Acknowledgments

We thank the anonymous reviewers and our shepherd Swami Sundararaman for their helpful comments and feedback. We thank Íñigo Goiri for providing part of the cloud traces for our study as well as insightful discussions. This work is supported by NSF CAREER Award 2144796, CCF-1919044, CNS-1850317 and a grant from Western Digital Technologies, Inc.

References

[1] Ahmed Abulila, Vikram S Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. FlatFlash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, Providence, RI, USA, 2019.

[2] Albumentations Image Processing. https://github.com/albumentations-team/albumentations, 2021.

[3] Alibaba Cluster Trace. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.

[4] Amazon Elastic Compute Cloud. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.

[5] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, November 2020.

[6] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, October 2014.

[7] Amazon AWS. Burstable performance instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html, 2020.

[8] Microsoft Azure. Introducing B-Series, our new burstable VM size. https://azure.microsoft.com/en-us/blog/introducing-b-series-our-new-burstable-vm-size/, 2017.

[9] Azure cloud trace. https://github.com/Azure/AzurePublicDataset, 2019.

[10] Ricardo Bianchini, Marcus Fontoura, Eli Cortez, Anand Bonde, Alexandre Muzio, Ana-Maria Constantin, Thomas Moscibroda, Gabriel Magalhaes, Girish Bablani, and Mark Russinovich. Toward ml-centric cloud platforms. *Communication of ACM*, 63(2), January 2020.

[11] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Santa Clara, CA, February 2017.

[12] Amazon Elastic Compute Cloud, Burstable Performance Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html, 2020.

[13] Microsoft Azure Cloud. Configure online virtual hard disk resize. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/dn282284(v=ws.11), 2016.

[14] Cloud flash storage: SSD options from AWS, Azure, and GCP. https://www.computerweekly.com/feature/Cloud-flash-storage-SSD-options-from\-AWS-Azure-and-GCP, 2020.

[15] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, 2017.

[16] Project denali to define flexible ssds for cloud-scale applications. https://azure.microsoft.com/en-us/blog/project-denali-to-define-flexible\-ssds-for-cloud-scale-applications/.

[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*, 2009.

[18] Andreas E Dilger. Online ext2 and ext3 filesystem resizing. In *Ottawa Linux Symposium*, page 117, 2002.

[19] Ev3 and Esv3-Series. https://docs.microsoft.com/en-us/azure/virtual-machines/ev3-esv3-series, 2021.

[20] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting vms in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, Lausanne, Switzerland, February 2022.

[21] Google. Our data centers now work harder when the sun shines and wind blows. https://blog.google/inside-google/infrastructure/data-centers-work-harder-sun-shines-wind-blows, 2020.

[22] Google Cluster Trace. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.

[23] Google Cloud. Preemptible VM Instances. https://cloud.google.com/compute/docs/instances/preemptible.

[24] Graphchi. https://github.com/GraphChi/graphchi-cpp, 2021.

[25] Hadoop TeraSort. https://hadoop.apache.org/docs/r3.2.0/api/org/apache/hadoop/examples/terasort/package-summary.html, 2021.

[26] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, November 2020.

[27] Timothy Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC 2001)*, Lisbon, Portugal, 2001.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[29] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Santa Clara, CA, February 2017.

[30] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified Address Translation for Memory-Mapped SSD with FlashMap. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA'15)*, Portland, OR, June 2015.

[31] IBM. Ibm flash storage and software defined storage. *White Paper*, 2017.

[32] Giorgos Kappes and Stergios V. Anastasiadis. Libservices: Dynamic storage provisioning for multitenant i/o isolation. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*, Tsukuba, Japan, 2020.

[33] Giorgos Kappes and Stergios V. Anastasiadis. A user-level toolkit for storage i/o isolation on multitenant hosts. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*, Virtual Event, USA, 2020.

[34] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *Proc. FAST'15*, Santa Clara, CA, February 2015.

[35] Shine Kim, Jonghyun Bae, Hakbeom Jang, Wenjing Jin, Jeonghun Gong, Seungyeon Lee, Tae Jun Ham, and Jae W. Lee. Practical erase suspension for modern low-latency SSDs. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*, Renton, WA, July 2019.

[36] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[37] Daniar H. Kurniawan, Levent Toksoz, Anirudh Badam, Tim Emami, Sandeep Madireddy, Robert B. Ross, Henry Hoffmann, and Haryadi S. Gunawi. Ionet: Towards an open machine learning training ground for i/o performance prediction. *Technical Report*, 2021.

[38] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. FVM: Fpga-assisted virtual device emulation for fast, scalable, and flexible storage virtualization. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, November 2020.

[39] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, Santa Clara, CA, February 2016.

[40] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale disaggregated storage. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)*, Santa Clara, CA, July 2017.

[41] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, Amsterdam, The Netherlands, 2014.

[42] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*, London, United Kingdom, April 2016.

[43] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, Portland, OR, June 2015.

[44] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[45] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, Beijing, China, 2019.

[46] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Carlsbad, CA, October 2018. USENIX Association.

[47] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'21)*, Virtual Event, USA, 2021.

[48] Pulkit A. Misra, Inigo Goiri, Jason Kace, and Ricardo Bianchini. Scaling distributed file systems in resource-harvesting datacenters. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC'17)*, Santa Clara, CA, July 2017.

[49] Dushyanth Narayanan, Eno Thereska, Austin Donelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds, analysis of tradeoffs. In *Proceedings of the Fourth European Conference on Computer Systems (EuroSys'09)*, Nuremberg, Germany, March 2009.

[50] Nutanix Distributed Storage. https://www.nutanix.com/products/acropolis/distributed-storage, 2022.

[51] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, Salt Lake City, UT, 2014.

[52] PyTorch. https://pytorch.org/, 2021.

[53] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, November 2020.

[54] RocksDB. https://github.com/facebook/rocksdb, 2021.

[55] Software-defined data center. https://en.wikipedia.org/wiki/Software-defined_data_center.

[56] Software-defined storage. https://en.wikipedia.org/wiki/Software-defined_storage.

[57] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Carlsbad, CA, October 2018.

[58] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In *Proceedings of the Fourteenth European Conference on Computer Systems (EuroSys'19)*, Dresden, Germany, 2019.

[59] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, 2015.

[60] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, October 2012.

[61] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnam Sundama, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, Renato Ferreira, Mohamed Nassar, Michael Koltachev, Ji Huang, Sudipta Sengupta, Justin Levandoski, and David Lomet. Schema-agnostic indexing with azure documentdb. In *Proceedings of the 41st International Conference on Very Large Databases (VLDB'15)*, Kohala Coast, Hawaii, September 2015.

[62] Azure Spot VM. https://azure.microsoft.com/en-us/services/virtual-machines/spot/.

[63] Error Messages for Azure Spot Virtual Machines and Scale Sets. https://docs.microsoft.com/en-us/azure/virtual-machines/error-codes-spot.

[64] Software-Enabled Flash for Hyperscale Data Centers. https://searchstorage.techtarget.com/post/Software-Enabled-Flash-for-Hyperscale\-Data-Centers, 2021.

[65] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*, 2013.

[66] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15)*, Bordeaux, France, 2015.

[67] VMware. Growing, thinning, and shrinking virtual disks in esxi. https://kb.vmware.com/s/article/1002019, 2021.

[68] VMWare VSAN. https://www.vmware.com/products/vsan.html, 2022.

[69] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: Harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21)*, 2021.

[70] Guanying Wu and Xubin He. Reducing SSD read latency via NAND flash program and erase suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, San Jose, CA, February 2012.

[71] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*, Tel-Aviv, Israel, 2013.

[72] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *arXiv preprint arXiv:1205.6233*, 2012.

[73] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/wiki, 2021.

[74] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. RLScheduler: An Automated HPC Batch Job Scheduler Using Reinforcement Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*, Virtual Event, November 2020.

[75] Ning Zhang, Junichi Tatemura, Jignesh M. Patel, and Hakan Hacigumus. Re-evaluating Designs for Multi-Tenant OLTP Workloads on SSD-based I/O Subsystems. In *Proceedings of the SIGMOD'14*, Snowbird, UT, June 2014.

[76] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, November 2016.

[77] Zhiheng Zhong, Minxian Xu, Maria Alejandra Rodriguez, Chengzhong Xu, and Rajkumar Buyya. Machine learning-based orchestration of containers: A taxonomy and future directions. *Computing Research Repository (CoRR)*, abs/2106.12739, 2021.

[78] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. In *Proceedings of the Machine Learning and Systems (MLSys'21)*, Austin, TX, March 2021.

MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime

Chenxi Wang[†♣] Haoran Ma[†♣] Shi Liu[†] Yifan Qiao[†] Jonathan Eyolfson[†] Christian Navasca[†]
Shan Lu[‡] Guoqing Harry Xu[†]
University of California, Los Angeles[†] *University of Chicago*[‡]

Abstract

Far-memory techniques that enable applications to use remote memory are increasingly appealing in modern data centers, supporting applications' large memory footprint and improving machines' resource utilization. Unfortunately, most far-memory techniques focus on OS-level optimizations and are agnostic to managed runtimes and garbage collections (GC) underneath applications written in high-level languages. With different object-access patterns from applications, GC can severely interfere with existing far-memory techniques, breaking remote memory prefetching algorithms and causing severe local-memory misses.

We developed MemLiner, a runtime technique that improves the performance of far-memory systems by "lining up" memory accesses from the application and the GC so that they follow similar memory access paths, thereby (1) reducing the local-memory working set and (2) improving remote-memory prefetching through simplified memory access patterns. We implemented MemLiner in two widely-used GCs in OpenJDK: G1 and Shenandoah. Our evaluation with a range of widely-deployed cloud systems shows MemLiner improves applications' end-to-end performance by up to **2.5**×.

1 Introduction

Datacenters are becoming increasingly *memory constrained* [65, 45, 40] with the ubiquitous deployment of in-memory data analytics and ML systems like Neo4j [52], Cassandra [12], Spark [74] and TensorFlow [5], which hold large amounts of intermediate data in memory for quick processing. To tackle this constraint, far-memory techniques [30, 10, 63, 58, 26] that enable applications to use remote memory are increasingly appealing, backed by advances in hardware and networking techniques [13, 62, 66, 23, 19, 28, 35, 49, 55, 59, 32, 8, 16, 38, 41, 33, 63, 43, 57, 37, 42, 60, 7] that allow remote memory to offer much lower latency and higher bandwidth than local block devices.

♣ Contributed equally.

Most of these far-memory systems [30, 10, 63, 48, 68] build on a cache-and-swap mechanism: the application's host server uses local memory as a *data cache*. Once a page that does not reside in the local memory is accessed, a page fault is triggered and the page is fetched from a remote server into the local memory. Good locality and effective remote-memory prefetching [50, 48] are crucial to the performance of applications running in such far-memory systems.

Unfortunately, the interference from garbage collection (GC) severely degrades the memory-access locality and remote-memory prefetching for applications written in high-level languages (e.g., Java, Go, and Python), which are dominant in datacenter workloads. At run time, application threads access heap objects following their program-execution paths, while GC threads *concurrently* scan the heap, performing graph traversal from a set of "roots" (*i.e.*, objects referenced by stack and global variables) to mark live objects. Object accesses by these two sets of threads are uncoordinated, creating two disjoint working sets, as illustrated by Figure 1(a), and causing severe performance problems.

Problem 1: Resource Competition. Pages swapped in for GC's heap traversal are often not used (in near future) and hence evicted by the application; conversely, pages swapped in for the application are often not needed (in near future) and evicted by GC. Evicting each other's pages, the application and GC both suffer from severe local-memory misses and further compete for RDMA bandwidth for page swapping. The more concurrent activities a GC runs , the more the resource competition between GC and the application—our results show that running Spark with the Shenandoah concurrent GC [25] on the 25% memory configuration incurs a 12× slowdown to the end-to-end performance, which is 5× larger than the default G1 GC that reclaims memory in stop-the-world pauses.

Problem 2: Ineffective Prefetching. Monitoring the execution of a managed program, an OS-level prefetcher such as [48] cannot recognize clear memory-access patterns and has to give up prefetching. The reason is that, even if the appli-

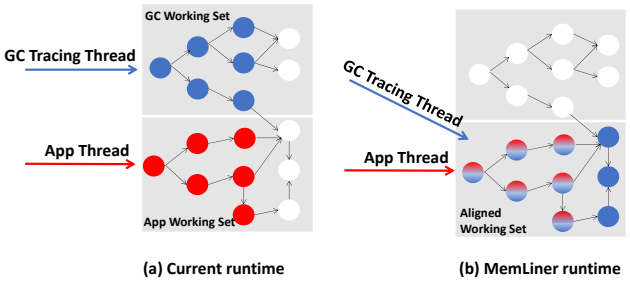


Figure 1: Our main idea: the working sets of GC threads, in blue, and application threads, in red, during a time window (a) without or (b) with the access alignment from MemLiner.

cation's memory accesses follow a simple sequential pattern, the combined accesses from both the application and the GC often appear random from the OS' perspective.

State of the Art. In the past, supporting applications that have large memory footprints (*e.g.*, larger than the main memory size) is not the priority of traditional GC. Although there exists a body of work (such as Platinum [70]) on concurrent GC, such work focuses primarily on improving throughput and reducing latency on memory-abundant servers. However, remote memory is designed to enable applications to use more memory than what their hosts can offer; as a result, developing new GC techniques to support these applications becomes a crucial task.

Recent work Semeru [68] supports running Java programs on disaggregated hardware by disaggregating the traditional JVM into two new ones, with the CPU-JVM executing the program on the CPU server and the memory-JVM performing GC on the memory server. The idea of offloading GC completely to a remote server works for Semeru where *all* the application's memory data is located in a remote server, but does not suit today's datacenters where resources are not entirely disaggregated and applications use remote memory only if their local memory runs out. Furthermore, this offloading approach imposes extra communication overhead for CPU-JVM and memory-JVM to coordinate, and extra computation cost on the remote memory server to run the memory-JVM, which may impose deployment challenges.

Another recent work AIFM [58] proposes a novel runtime to improve the prefetching and swap performance of applications running in remote-memory systems. AIFM targets applications written in native languages (C/C++), and hence cannot easily be applied to solve the GC interference problem in the managed language runtime.

MemLiner. This paper presents a fully-automated runtime technique, MemLiner, for programs written in high-level languages (HLLs) to efficiently use remote memory.

The design of MemLiner is based on two key observations.

First, the objects accessed by the application and the GC are not completely unrelated—they are just not temporally aligned. The live objects traced by the GC are mostly accessed by the application at some point during the execution; the objects accessed by the application must be live objects at the moment of the access and hence the target of GC.

Second, although changing object-access order in application threads would break the application semantics, changing that order in GC would not. Specifically, GC threads aim to trace and mark all reachable objects in the heap, while the *order* of that tracing and marking (*e.g.*, which objects are traced first) does not matter.

Guided by these observations, the key idea behind MemLiner is *working set alignment*. MemLiner carefully reorders the objects traced by the GC threads, so that they follow a similar, although not identical, memory-access path of the concurrent application threads (illustrated by Figure 1(b)). Consequently, their working sets can better overlap with each other; the resource competition can be much alleviated, with much reduced page faults and on-demand swaps; the application's access patterns can be more easily recognized by the underlying prefetcher such as Leap [48]. All of these are achieved in a way that is compatible with existing GC algorithms, without offloading the GC to another machine or re-desgining the prefetcher.

MemLiner must overcome several challenges.

First, *how to align GC threads with application threads*. In a conventional setting, GC traces objects using a graph traversal starting at the root objects. To align GC's accesses with application threads', MemLiner uses a *priority-based* algorithm—MemLiner makes application threads inform the GC of the objects they are accessing; these objects, which must be live and reachable in the object graph at that moment, are then immediately traced and marked by the GC, without any risk of triggering page faults and expensive remote swaps. To enable such communication, MemLiner leverages the *read-write barrier*—a piece of code executed by the runtime at each heap read/write in the application—to inform GC of the objects on the application's access path. Details of the coordination are discussed in §4.1.

Second, *when to break the alignment* so that GC can finish its work without unnecessary delays. Completely aligning GC threads with application threads could severely delay GC from reclaiming dead heap space, as application threads may take a long time, sometimes even the whole execution, to access every live object. In fact, a complete alignment is unnecessary, as application threads may repeatedly access the same object in a short time window due to application semantics, like during a loop, while GC only needs to mark that object live once. Consequently, MemLiner allows GC to break from the alignment to work on another part of the heap traversal from time to time. To minimize the interference, MemLiner prioritizes two types of objects in GC's unaligned accesses: (1) objects that will likely be accessed by the application soon; (2) objects that were accessed by the application not long ago and hence are likely still inside the local memory. The former is predicted based on what objects

the application just accessed; the latter is predicted based on object-access history that MemLiner efficiently encodes inside the per-object pointer. Details can be found in §4.2.

Results. We have integrated MemLiner into two widely used GCs (G1 and Shenandoah) in OpenJDK 12. A thorough evaluation with Spark, Cassandra, Neo4J, QuickCached and DayTrade demonstrates that MemLiner improves the end-to-end execution time by an overall of **1.48**× and **1.51**× under the 25% and 13% local memory configurations for the G1 GC, and **2.16**× and **1.80**× for the Shenandoah GC (which runs concurrent GC threads more frequently than G1). Furthermore, MemLiner improves Leap's prefetching coverage and accuracy by **1.5**× and **1.7**×, respectively. Compared to Semeru [68], MemLiner achieves a comparable performance without offloading any computation on remote servers.

Key Takeway. Although there are several directions of work on remote memory (*e.g.*, clean-slate approaches such as AIFM [58] and Kona [17], swap optimizations such as InfiniSwap [30] and FastSwap [10], as well as distributed runtimes such as Semeru [68]), MemLiner takes an *easy-to-adopt, non-intrusive* approach that enables performance improvements for a wide variety of new and legacy applications. MemLiner is orthogonal to (and complements) these existing techniques—aligning the memory accesses between application and GC threads reduces thread-level interference and the application's local-memory working set regardless of the underlying remote-access mechanisms and optimizations.

2 Background

GC. A major benefit of high-level languages over native languages is their support for automated memory management—developers are released from the burden of deallocating objects, leading to improved reliability and security. Automated memory management is enabled by garbage collection (GC), which runs when the heap has little free space. The key idea of GC is simple [36]: perform a reachability analysis to identify a transitive closure of live objects and reclaim objects outside the closure. Consequently, a modern GC algorithm has two main components: (1) *tracing* the heap graph to compute that closure and identify live objects, and (2) *reclamation* of dead objects, while evacuating live objects to contiguous space and updating pointers.

Concurrent Tracing. To ensure the correctness of pointer updating, a conservative way of running GC is to pause all application threads (*i.e.*, a stop-the-world phase) for full-heap tracing and reclamation, which incurs significant delays [53, 47]. To address this performance limitation, starting from the G1 GC [22], which is the default GC in Oracle's JVM, all modern garbage collectors, including Shenandoah [25] from Red Hat and ZGC [2] from Oracle, run the tracing phase concurrently with application threads to (1) leverage the many available cores and (2) minimize GC pauses. For example, in G1, the number of tracing threads is configured, by default, to be 1/4 of the number of cores. Concurrent tracing often uses a snapshot-at-the-beginning (SATB) algorithm [73]—tracing traverses the heap graph from a logical snapshot of the heap; it will not miss any live object as long as object allocation and pointer updates made by the application since the snapshot are recorded and considered conservatively. G1 runs stop-the-world phases to reclaim memory by evacuating live objects into new regions while Shenandoah and ZGC run evacuation also concurrently to minimize the pause time.

Tracing Algorithm. Logically, tracing divides objects into three colors: *white*, *black*, and *gray*. The white set is the set of objects that are candidates for reclamation. The black set is the set of objects that can be shown to have no references going to objects in the white set, and to be reachable from the roots. Objects in the black set are not candidates for reclamation. The gray set contains all objects reachable from the roots but yet to be scanned.

Initially, all objects are white. Tracing implements a graph traversal algorithm that gradually changes the color of objects reachable from the roots from white to black. For each reachable object *o*, tracing marks it black, retrieves all objects referenced directly by *o*, and adds them into the gray set. Each iteration retrieves an object from the gray set, marks it black, and adds more objects into the gray set. The algorithm repeats until the gray set becomes empty; objects that remain white can be safely reclaimed. In practice, a modern runtime uses a bitmap to mark live objects efficiently.

3 Motivation

In this section, we use an experiment to quantitatively demonstrate (1) how tracing and application threads interfere with each other, and (2) why simply disabling concurrent tracing cannot solve the problem.

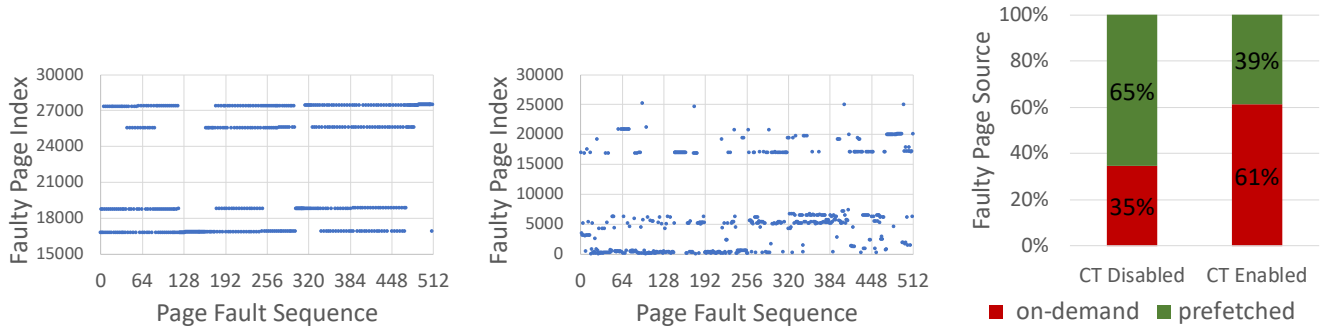
Setup. We ran Spark Logistic Regression (LR) with the Wikipedia dataset on OpenJDK 12 and its default G1 GC. We used two machines, each with 2 Xeon(R) CPU E5-2640 v3 processors, 128GB memory, 1024GB SSD, and CentOS 7.5, connected by RDMA over a 40Gbps InfiniBand network. One machine runs Spark, using local memory and remote memory on the other machine. We configured the first machine to have just enough memory to host 25% of Spark's working set. We name the first server providing compute resource as *host server* and the second server providing remote memory as *remote server*.

We compare the execution of Spark LR in two modes:

(1) The G1 GC's concurrent tracing is *disabled*;

(2) The G1 GC's concurrent tracing is *enabled*—the default option in G1 GC. The number of tracing threads is set to be a quarter of the number of available cores, as suggested by G1.

In both cases, the heap size of Spark LR is set to 32GB and the host server can hold up to 8GB of its heap. The execution goes through application-execution phases and stop-the-world GC phases alternatively.



(a) Page fault trace of Spark LR. (b) Page fault trace of Spark LR with CT. (c) Prefetching effectiveness.

Figure 2: Prefetching effectiveness for Spark LR executed atop OpenJDK 12 (with its default G1 GC): (a) trace of faulty page index for application threads only; (b) trace of faulty page index when concurrent tracing (CT) is enabled; (c) disabling CT significantly improves the effectiveness of Linux' default swap prefetcher.

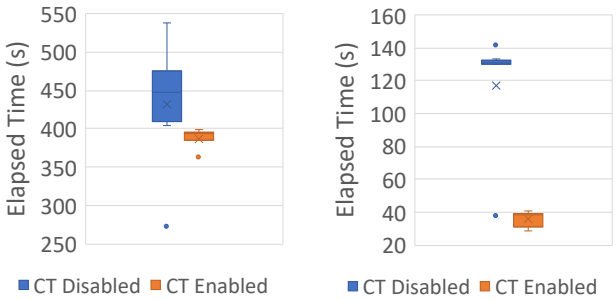
How much interference from concurrent tracing? To have an intuitive look at how well prefetching may or may not work, we randomly sampled 512 *consecutive* page faults in the middle of Spark LR's execution under both execution modes. Note that, since we collected page-fault information from inside the kernel and the execution under the two GC modes proceeds at vastly different paces, we cannot guarantee that the two samples come from the same window of application instructions, but we do make sure that the stop-the-world GCs did not occur during our samples.

Figure 2 (a) and (b) illustrate the virtual page index of the faulty addresses (Y-axis) ordered by when each fault occurs, with the sequence number shown in the X-axis. Without concurrent tracing, each of the application threads has a clear streaming access pattern, as shown in Figure 2(a), which should be detected by an advanced prefetcher. This clear pattern is messed up by concurrent tracing, as shown in Figure 2(b), making prefetching much harder.

To quantitatively measure the impact of concurrent tracing on prefetching, we checked 500 application-execution phases (*i.e.*, the period between two stop-the-world GCs) to understand, among all the page faults, how many were resolved through on-demand swaps from remote memory and how many were resolved using data already brought in through prefetching. Clearly, this ratio of on-demand swapping versus prefetching directly affects the application performance.

As shown in Figure 2(c), without concurrent tracing, prefetching is effective, addressing 65% of the page faults. Unfortunately, with concurrent tracing, this ratio greatly dropped to only 39%, with the remaining 61% of page faults leading to costly remote-memory accesses. Note that our experiments use Linux's default swap prefetcher. If an advanced prefetcher such as Leap [48] is used, the prefetch-ratio would be even higher without concurrent tracing and hence suffer even more from the interference (see §7).

Finally, to understand how much the interference has affected the working set of the execution, we also measured the average number of page faults encountered by application threads. The page-fault rate jumps from **3.5K** per second per thread to **9.6K** per second per thread, when concurrent tracing is enabled, indicating a huge interference.



(a) End-to-end execution time. (b) GC pause time.

Figure 3: Concurrent tracing improves overall performance. (Data is from 10 runs of each program; dots are outliers.)

Why not just disable concurrent tracing? Having seen significant interference from concurrent tracing, a strawman solution is to simply disable concurrent tracing for applications running in far-memory systems.

Unfortunately, this strawman solution does not work. First, modern concurrent GCs such as Shenandoah [25] and ZGC [2], which are designed for low-pause and used widely by latency-sensitive cloud applications, rely on concurrent tracing to reclaim memory (also concurrently). Disabling concurrent tracing would destroy the functionality of such collectors. Second, even for GCs such as G1 that could perform tracing in a stop-the-world phase, the end-to-end execution time suffers significantly without concurrent tracing. As shown in Figure 3(a), the execution time increases by **18%** on average in 10 runs. The main reason is that the aggregated stop-the-world GC periods now take **2.7**× longer without concurrent tracing, as shown in Figure 3(b). Without concurrent tracing, each (fast young-generation) GC cannot

reclaim as many dead objects in the same amount of time and has to resort to *slow, full-heap GC* that scans and compacts the whole heap space in a stop-the-world period, which is extremely time consuming. For example, the longest full-heap GC (*i.e.*, a single pause) in Spark LR takes 76.9 seconds, clearly an intolerable delay.

Key Takeway. Memory accesses from application and GC threads exhibit diverse patterns, significantly increasing the application's working set and making prefetching harder. Simply disabling concurrent tracing in GC would not work, as it reduces the number of local-memory misses at a cost of significantly increased GC pause and end-to-end execution time. MemLiner offers a solution that can greatly reduce the number of local-memory misses and increase the effectiveness of existing prefetchers without introducing extra GC-pause time, and hence effectively reduce the end-to-end execution time.

4 MemLiner Design and Implementation

This section presents the design and implementation of MemLiner, particularly how we realize the two key ideas: (1) making GC concurrently trace objects immediately after their access by application threads (§4.1) and (2) making GC trace other live objects through a novel priority-based algorithm (§4.2) to reduce interference.

MemLiner modifies the garbage collector inside the runtime and the swapping system inside the kernel, while requiring *no* changes to applications. In terms of runtime changes, MemLiner is a general mechanism that can be integrated into any modern runtime that performs concurrent tracing. This paper focuses on a design for Oracle's OpenJDK, a commercial JVM that supports a variety of high-level languages such as Java, Scala, Python, Ruby, *etc.* In terms of kernel changes, we build MemLiner atop paging/swap mechanisms that already exist in the OS kernel, with minimal invasion. Any swap optimizations such as InfiniSwap [30] and FastSwap [10] can be readily used to improve the swap performance for a MemLiner-equipped runtime. MemLiner's runtime design is independent of how remote memory is accessed; for example, MemLiner could also run on a clean-slate platform such as Kona [17] that access remote memory based on cache coherence, not page faults, if coherence is provided by hardware.

When a MemLiner-equipped JVM is launched, the maximum heap size M is specified by the user via a command-line option. A small amount of physical memory on the local machine is initially used to back up the heap (which is much smaller than M). The heap stays entirely in local memory until its usage exceeds the size of local memory, in which case, the OS kernel allocates remote memory by registering it as an RDMA buffer. The kernel uses an approximate LRU algorithm to evict pages. MemLiner does not require any software/hardware support on remote servers, providing a practical solution that can be readily used in today's cloud.

4.1 Application and GC Coordination

To align memory accesses, application threads inform GC's tracing threads of the objects they are accessing so that tracing threads can trace these objects immediately.

To facilitate such communication, we need to instrument every heap read/write instruction so that the application can send an object pointer to GC when it dereferences the pointer: *(1)* At a statement that reads an object field or an array element of the form $a = b.f$ or $a = b[i]$, our instrumentation pushes the corresponding address in b into a thread-local producer-consumer queue (PQ), which will be read by GC during tracing. *(2)* At a statement that writes an object field or an array element of the form $b.f = a$ or $b[i] = a$, we similarly push the object reference in b into the PQ.

MemLiner implements this instrumentation through existing read/write barriers—a piece of code that is executed by modern runtimes at each heap read/write operation to record heap information for GC purposes. MemLiner piggybacks on the existing implementation of read/write barrier in OpenJDK 12 that intercepts both interpreted and compiled code. A PQ is created for each application (producer) thread so that no synchronization is needed for enqueuing pointers. A GC tracing (consumer) thread constantly checks PQs to retrieve pointers for tracing. Consumer threads use atomic instructions when dequeuing object pointers. In practice, the number of application threads is often larger than the number of tracing threads; hence, there is little contention when PQs are accessed by multiple threads.

To minimize the maintenance overhead, we represent each PQ as a non-blocking *ring buffer*. Producers and consumers do not synchronize at all—an application thread keeps writing into the queue even if it is full. As such, the application thread may overwrite entries that have not yet been picked up by GC. Note that this would not cause any correctness issues because those entries only indicate *tracing priority*: overwriting an entry will delay the corresponding object's tracing, but the tracing of these objects will eventually happen in GC's regular graph traversal, which will be discussed in the next sub-section.

Note that our instrumentation code at different program points is unlikely to enqueue the same object reference multiple times (*e.g.*, neighboring reads to the same data structure). This is because marking an object live sets a bit in a global live bitmap. Before pushing each object reference into the queue, an application thread checks its bit from the bitmap and filters it out if the bit is already set.

4.2 MemLiner Tracing Algorithm

4.2.1 Design Overview

A major challenge in aligning tracing and application threads is that GC has to compute a *full* closure of live objects to reclaim memory. Hence, it is unproductive to trace a live object only right after it is accessed by the application, which

will delay the closure computing, leading to inefficiencies in memory reclamation.

The key question here is: *how can GC make quick progress in closure computation without producing a working set that significantly departs from that of the application?* On the one hand, after processing all objects in the PQ, we want GC to trace as many other live objects as possible, even if not in the PQ, to complete the closure. On the other hand, GC should better *not* trace many objects that do not reside in local memory because tracing those objects triggers page faults and swaps. How to reconcile these seemingly conflicting goals is a problem MemLiner must solve.

Reachable Object Classification. To better explain our tracing algorithm, we first classify all live objects at any moment of the execution into three categories based on their location and when they are accessed by the application, as illustrated in Figure 4[1]:

*(1) Objects in local memory (*i.e.*, data cache):* These objects have recently been accessed by the application and have not been evicted yet. Clearly, tracing them at this moment (or in the near future) would not generate any page faults or interfere with the application. Many of these object (*i.e.*, the red ones in the figure) are made known to the GC through the PQ discussed in §4.1. However, since the PQ is designed to be a ring buffer, some of these objects (*i.e.*, the striped ones in the figure) may be missed by GC due to being overwritten in the ring buffer. How to trace them sooner rather than later requires extra handling that we will discuss later.

(2) Objects in remote memory and to be used soon: Since these objects (*i.e.*, the wavy nodes in Figure 4) will soon be accessed by the application, they are typically just a few references away from the objects being accessed by the application. Tracing them is also desirable—although they are currently not local, they will soon be needed by the application. If GC triggers page faults when accessing them, the costs of handling these faults and swapping would be *necessary* as they are "prepaid" by GC for the application.

(3) Objects in remote memory and not used soon: These are illustrated as clear-circle objects in the figure. They were used by the application a while ago and got evicted to remote memory. Tracing them is needed *eventually* but is undesirable now or in the near future, as tracing them pays the high cost of fault handling and swapping (which is entirely wasted if they are not used by the application before their next eviction).

Handling Different Categories in GC. MemLiner's central design goal is to let GC trace objects in Category (1) and (2) right away *to maximize progress* and delay tracing objects in Category (3) *to avoid unnecessary page faults and interference*. Among the different categories of objects, our starting point is the set of red objects, which are captured by the read/write barrier, sent to GC via the PQ, and traced by GC immediately.

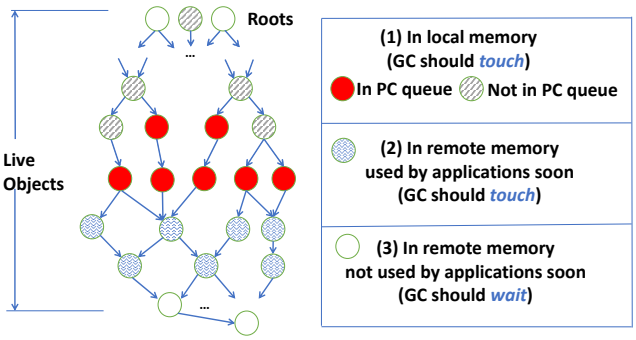


Figure 4: Classification of reachable objects in the heap: red objects are being accessed by the application and shaded objects are what MemLiner intends to trace.

With the red objects in hand, the wavy objects in Category (2) are just a few references away. To mark these objects, we let GC trace *a small number of references forward* from the red objects, which were retrieved from the PQs. As discussed above, tracing such an object will likely trigger swapping, prepaying the cost for the application to access the object soon later. Note that tracing too many references forward will not be useful, as that may bring in objects not used by the application in the near future. In our implementation, we limit the number of hops to 3, which is often large enough to cover objects in the same logical data structure [72].

After red objects and wavy objects, the remaining live objects to trace are those in Category (3) and the striped objects in Category (1). There are two challenges here. First, there are no easy ways to reach them from the red objects. Second, to reduce memory interference, it is better to trace the striped Category (1) objects before the Category (3) objects, as discussed above.

To tackle these challenges, MemLiner makes every concurrent tracing thread alternate between two modes:

(1) When the PQ is not empty, trace objects in the PQ (*i.e.*, red) and objects a few references forward (*i.e.*, wavy);

(2) When the PQ is empty, perform normal object-graph traversal that starts from root objects like traditional GC.

Different from a traditional GC, MemLiner modifies the traversal algorithm to consider whether an object o to be traced is likely in local memory (*i.e.*, whether o is a striped Category (1) object or a Category (3) object)—if o is *estimated* to reside in local memory (*i.e.*, a striped Category (1) object), it is traced right away in GC; if not (*i.e.*, Category (3)), MemLiner postpones processing o in its graph traversal until a later time, optimistically hoping that o will be used by the application before it is encountered again in GC. After postponing a number of times (referred to as MAX_DL below) , GC processes o even if it is still estimated to be remote, so that the closure computation will not be significantly delayed. MemLiner dynamically adjusts the value of

[1]For ease of discussion, here we do not consider cold objects staying in cache due to hot objects on the same page. We will discuss it in Section 4.3.

Figure 5: A 64-bit object pointer in MemLiner.

MAX_DL, in response to the size of available heap space. For example, when the available heap size is in the *red zone* (*i.e.*, <15% available space), MAX_DL will be set to 0, letting GC quickly finish tracing and collect memory. Details of this adaptive algorithm can be found in this section.

4.2.2 Object Location Estimation

Now, the only missing piece of MemLiner's tracing algorithm is a way to *estimate* whether an object is local or not. A naïve solution is to create a system call that allows GC to query the page table. However, this can be prohibitively expensive as it requires a system call per object visited during tracing.

To solve this problem, we conceptually divide the execution into *epochs* and encode the current epoch ID into each *object pointer* whenever an object is accessed. Later on, during concurrent tracing, this epoch ID will allow the GC to estimate how recently an object was accessed and hence how likely it is still in local memory.

Epoch. Given our goal of estimating whether an object is in local memory, we define an *epoch* to be an execution period in which the set of pages in local memory that belongs to the JVM process are *relatively stable* (*i.e.*, they do not change much). This set changes as new pages of this JVM process are swapped in and old pages are swapped out. When the change becomes significant (*e.g.*, larger than $N\%$ of the total number of JVM pages), a new *epoch* starts. We modify the kernel swap system to keep track of the pages in the cache and determine the start of a new epoch. A global epoch counter is maintained in the JVM and its address is passed into the swap system. This epoch counter starts from zero and is increased by one whenever a new epoch starts.

Timestamp. In the JVM, virtual addresses of objects are represented as *references*, which are essentially pointers with a strong type. In a 64-bit JVM, the format of an object reference is shown in Figure 5. Recall that our need is to estimate whether an object is in local memory from a reference/pointer of the object (*e.g.*, recorded in a field of another object) during GC's graph traversal. Our idea here is to modify the pointer format by reserving 4 unused bits as a *timestamp* (ts in Figure 5) that indicates *the epoch in which the pointer was last dereferenced*—once the epoch ID reaches 15, the next epoch ID goes back to 0. Dereferencing the pointer accesses the target object (*i.e.*, bringing the object to local memory if it is remote). As such, if the timestamp is close to the current epoch, the object is likely in local memory (*i.e.*, Category (1)) and GC should follow the pointer to trace the object; otherwise, the object may not be local (*i.e.*, Category (3)), and GC should postpone tracing it.

Algorithm 1: Allocation semantics.

Input: Allocation site $o = new\ C$.
Output: Object reference o.

```
1 addr ←ALLOCATE(SIZEOF(C))
2 o ←UPDATEPOINTER(addr, CURRENTEPOCH())
3 return o
```

Algorithm 2: Object read and write semantics in application threads.

Input: Object read/write access $a = b.f$ or $b.f = a$.

```
1 ENQUEUE(PQ, b)
2 b ← UPDATEPOINTER(b, CURRENTEPOCH())
3 if ISREFERENCE(a) then
4     b.f ← a ← UPDATEPOINTER(a, CURRENTEPOCH())
```

Upon the allocation of a new object o, MemLiner sets the timestamp bits in o's pointer to be the current epoch number (with function UPDATEPOINTER in Algorithm 1).

Whenever an object is read/written in an application thread like $b.f = a$ or $a = b.f$ (Algorithm 2), MemLiner updates the timestamp ts in the dereferenced pointer b to be the current epoch ID. Furthermore, if a and $b.f$ are also object references, we write an updated pointer of a into $b.f$, indicating that soon the object referenced by $b.f$ will be accessed through a. Again, this instrumentation is implemented through read-/write barriers.

Note that we use Algorithm 1 and Algorithm 2 to illustrate the high-level logic. Our implementation actually inserts assembly code for efficiency. Changing object pointers in the JVM would not cause problems for actual memory accesses—although each pointer represents a virtual address, the barriers we use mask pointers so that only the last 42 bits are used to access memory.

4.2.3 MemLiner Tracing Algorithm

Algorithm 3 shows GC's tracing logic, which was summarized in §4.2.1. The algorithm takes two queue data structures as input: TQ is a standard tracing queue (already used by the JVM) that contains references yet to be explored in object graph traversal; it is initialized with a set of object references in the stack and global variables (*i.e.*, roots). PQ, as discussed earlier, is the producer-consumer queue that contains references of red objects sent to GC by application threads.

As discussed in §4.2.1, every tracing thread of MemLiner alternates between two modes. In the default mode, tracing loops over the tracing queue TQ, shown in Line 2-13 in Algorithm 3, to perform normal graph traversal. Whenever PQ is not empty (Line 3), the tracing thread interrupts the normal traversal and switches to the other mode to handle the (red) objects in PQ (Line 4); this logic is listed in Algorithm 4 and will be discussed shortly.

Algorithm 3: Main tracing logic in MemLiner's GC.

Input: (1) Producer-consumer queue PQ; (2) tracing queue TQ.
Output: Fully marked live bitmap for all live objects.

```
1  Function TRACING(TQ, PQ):
2      while TQ ≠ ∅ do
3          if PQ ≠ ∅ then
4              TRACEREDANDCATEGORY2(TQ, PQ)
5          Tuple ⟨o, dl⟩ ← DEQUEUE(TQ)
6          if DIFF(TS(o), CURRENTEPOCH()) > δ ∧ dl <
            MAX_DL then
7              ENQUEUE(TQ, ⟨o, dl + 1⟩)
8              Continue
9          if CHECKLIVEBITMAP(o) = 0 then
10             MARKLIVEBITMAP(o)
11             foreach Non-null reference-type field f ∈ o do
12                 Object reference p ← o.f
13                 ENQUEUE(TQ, ⟨p, 0⟩)
```

Algorithm 4: Tracing logic for red and Category-(2) objects.

Input: (1) Producer-consumer queue PQ; (2) regular tracing queue TQ.

```
1  Function TRACEREDANDCATEGORY2(TQ, PQ):
2      while PQ ≠ ∅ do
3          o ← DEQUEUE(PQ)
4          EXPLORE(o, TQ, 0)
```

Input: (1) Object reference o; (2) tracing queue TQ; (3) current exploration depth $depth$.

```
5  Function EXPLORE(o, TQ, depth):
6      MARKLIVEBITMAP(o)
7      foreach Non-null reference-type field f ∈ o do
8          Object reference p ← o.f
9          if depth < MAX_Depth then
10             EXPLORE(p, TQ, depth + 1)
11         else
12             ENQUEUE(TQ, ⟨p, 0⟩)
```

In the default mode, each iteration of the tracing loop retrieves a 2-tuple $\langle o, dl \rangle$ from TQ, representing an object reference o and a *delay limit* dl. MemLiner compares TS(o) with the current epoch ID (Line 6). If these two IDs are close to each other (DIFF(TS(o), CURRENTEPOCH()) $\leq \delta$), MemLiner goes ahead to mark this object in the global live bitmap (Line 10) and pushes all the non-null object references stored in this object into the tracing queue TQ (Line 13). Otherwise, MemLiner estimates that the object is not in the cache and hence pushes this tuple back into TQ (Line 7), hoping that the application will use this object and bring it to the cache before the next time it is dequeued in tracing. To avoid pushing back an object too many times, which would delay the completion of closure computation, MemLiner uses a *delay limit* dl, which is initialized to 0. Every time a tuple is pushed back, its dl is incremented (Line 7). Once it becomes *MAX_DL* (*i.e.*, the additional check at Line 6), GC is forced to mark the object. *MAX_DL* is auto-tuned based on the amount of available heap space (discussed shortly).

The other mode of tracing red objects is triggered when PQ is not empty, as illustrated in Algorithm 4. Similar to the default tracing loop, each iteration of the loop (Line 2) in Algorithm 4 retrieves an object reference from PQ, calling a recursive function EXPLORE to not only mark red objects themselves, but also trace a few references forward to mark objects in Category (2), which may be soon used by the application. We use a recursive function here to control the number of references (*i.e.*, data structure depth) to be explored—once $depth$ exceeds a constant *MAX_Depth* (Line 9, 3 by default), the function does not further explore the object graph, but instead, pushes these unexplored references into the regular tracing queue TQ (Line 12) so that they can be traced later in a normal graph traversal without priority. This is because, as discussed in §4.2.1, following long reference chains can swap in objects that may not be needed by the application in the near future, leading to wasted efforts.

Marking an object live flips its corresponding bit in a global live bitmap (Line 6); as a result, the regular graph traversal (Algorithm 3) would not mark it again if it is encountered there. Once the tracing of the red and Category-(2) objects is done, GC resumes the normal graph traversal in Algorithm 3.

In modern GC with concurrent tracing, each tracing thread works on its own tracing queue TQ. MemLiner modifies each tracing thread to run Algorithm 3 so that the work on TQ is interrupted if there are outstanding red objects in a PQ. Each application thread independently pushes red objects into its thread-local PQ while each tracing thread can consume objects from all PQs. This design makes it possible to enable *work stealing* between threads to balance the number of red and Category-(2) objects processed by these threads. The read/write barrier is already used in existing GC algorithms, such as G1, Shenandoah and ZGC, as well as other far-memory techniques such as AIFM [58]. To further reduce MemLiner's overhead at each read/write barrier, we only need to push the object reference o (64 bit) onto the queue with a very small number of instructions.

Autotuning of MAX_DL. How much delay should be introduced to tracing depends on how urgently GC must be completed. As a result, we develop an autotuner that dynamically adjusts the value of *MAX_DL* in response to the available heap size. The rationale is straightforward: if the heap is almost full, there is an urgent need to complete GC and hence we should use a small value for *MAX_DL*; on the contrary, if the heap is mostly available, delaying GC will not have a large impact on memory and hence we use a large value for *MAX_DL* to minimize interference.

MemLiner uses two thresholds for heap availability: 15% and 50%. When the percentage of available memory is lower than 15%, the JVM is in a *red zone*. If the percentage is between 15% and 50%, it is in a *yellow zone*. The JVM is in a *green zone* if the amount of available memory is higher than 50% of the heap size. MemLiner monitors heap usage upon allocations and uses three values for *MAX_DL*: 0, 2, and 4 respectively if the heap falls in the red, yellow, and green zone. These thresholds were empirically chosen and worked well for all our applications.

4.3 Discussion

MemLiner performs adaptation in two dimensions: (1) adapting timestamps based on the swap behavior and (2) adapting *MAX_DL* based on heap availability. The swap behavior correlates with interference and heap availability correlates with GC urgency. We elaborate on how (1) and (2) work in harmony to make MemLiner achieve superior performance.

For (1), MemLiner uses the timestamp mechanism to reduce the interference between GC and application threads. For example, if the cached pages rarely change (*i.e.*, the application has excellent locality or the local memory size is large enough), the interference is minimal and hence it would not create performance issues if MemLiner does not deviate much from an existing GC. Indeed, our algorithm makes the global epoch change slowly and timestamps in most pointers are the same as the current epoch ID. Algorithm 3 would trace most objects in *TQ* without delays. This is a desired property—when resources are *not* constrained, MemLiner would not incur overhead because GC can trace objects and reclaim memory in a timely fashion.

Conversely, if the set of cached pages frequently changes (*i.e.*, the application has poor locality or the cache size is small), the interference is significant and MemLiner should perform differently from an existing GC. Indeed, the global epoch moves at a fast speed. As such, the timestamps in most pointers are different from the current epoch ID. In other words, most objects in the heap are Category-(3) objects that are not in local memory. Consequently, Algorithm 3 would delay the marking of most objects and thus make slow progress. This is also a desired property—tracing should "yield" to the application when local memory resource is tight and application threads are constantly accessing remote memory. In this case, MemLiner imposes a delay to GC, and the delay is bounded by *MAX_DL*.

For (2), we use heap availability to dynamically adjust *MAX_DL*, enabling MemLiner to "override" the policy made under (1) in urgent situations. For example, if the application is experiencing frequent changes in cached pages (indicating interference) while the heap is almost full, the policy under (1) would delay tracing, which can, in turn, delay the completion of GC and subsequently trigger an undesired full-heap collection. In this case, our adaptation under (2) would determine that the heap is in the red zone and thus change *MAX_DL* to 0—even if tracing is delayed, the delay length is set to 0, effectively allowing GC to move in a normal pace.

5 GC-Specific Optimizations

We have implemented MemLiner in both the JVM's default G1 GC [22] and Red Hat's Shenandoah GC [25], which are two representative GCs widely used in cloud settings. G1 is a generational GC that optimizes for throughput with stop-the-world pauses while Shenandoah is a concurrent GC that minimizes the time of each pause by concurrently tracing and compacting objects. Shenandoah optimizes for latency at the cost of reduced throughput. Our goal is to demonstrate that MemLiner can be easily integrated into both GC algorithms, providing performance benefits for different kinds of (*e.g.*, latency-sensitive or batching) workloads.

One challenge in MemLiner is its reliance on read and write barriers, which, if used naïvely, can incur a significant runtime overhead. This section discusses our optimizations to mitigate the overhead. With these optimizations, MemLiner's barrier introduces an average of 2% and 5% overheads, respectively, to Shenandoah and G1, when the application runs entirely with local memory. Such low overheads are due to the following reasons:

First, Shenandoah already utilizes both read and write barriers for concurrent tracing and concurrent evacuation. MemLiner only inserts few instructions into the existing barriers, incurring negligible overheads.

Second, the original G1 only uses the write barrier. Naïvely adding the read barrier into G1 can cause a much higher overhead. We develop the following three optimizations that successfully filter out a significant fraction of object accesses:

Optimization #1: The enqueue operation of MemLiner's barriers is enabled only when concurrent tracing is in progress. When concurrent tracing is not running, it is unnecessary to add any objects into the PQ.

Optimization #2: G1 is a generational GC that splits the heap into a young and an old generation. Concurrent tracing scans only *old-to-old references* (to compute garbage ratio for each region in the old-gen), meaning that references in the young generation are not traced in concurrent tracing at all. Based on this insight, our read barrier filters out all references in the young generation—there is no need to update their timestamps or add them in PQ because these references are not traced in G1's concurrent tracing anyways.

Optimization #3: Our read barrier does not need to update timestamps for objects whose pointer timestamp is the same as the epoch ID. Essentially, we use a check that first compares the pointer timestamp with the epoch ID and updates the timestamp only if they do not have the same value. The larger the local memory percentage is, the less frequently the epoch changes and hence more objects can benefit from this optimization. This explains why when the percentage of local memory increases, MemLiner's overhead does not increase proportionally (as shown in Figure 7).

6 Limitations

MemLiner is designed for managed applications running on a managed runtime and thus not applicable to native applications such as those written in C/C++. Furthermore, MemLiner is designed to optimize throughput (by reducing interference and improving prefetching), *not* latency. However, it does not increase the application latency (*i.e.*, making remote access longer) or the GC pause time. For the Shenandoah GC, its pauses are already very short because operations requiring a pause do not involve many remote accesses and their time is not changed much by MemLiner. For G1, by lining up the tracing and application's memory accesses, MemLiner makes concurrent tracing more efficient, thereby significantly reducing the frequency of triggering full-heap collections. However, it does not reduce the per-collection pause time.

As shown in our evaluation, the more remote memory an application uses, the more effective MemLiner's optimization. However, when a large percentage of the working set fits into local memory, MemLiner's effectiveness reduces. In fact, if this percentage exceeds 50%, MemLiner's performance is on par with that of the original JVM.

The other limitation is that MemLiner focuses on reducing interference between the application and concurrent tracing threads. Application threads may also interfere with memory reclamation threads if the GC performs concurrent reclamation (such as Shenandoah and ZGC). MemLiner cannot reduce this type of interference.

7 Evaluation

7.1 Experiment Setup

We implemented MemLiner on top of OpenJDK 12 (v 12.0.2) and Linux (v 5.4.0). Our swap system is based upon our re-implementation of FastSwap [10][2], which provides good swap performance. We implemented it on top of G1 and Shenandoah. Implementing MemLiner in other GCs would be straightforward in the future.

Environment. We ran our experiments with two machines, each with two Xeon(R) CPU E5-2640 v3 processors, 128GB memory, one 1TB SSD, and one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter. They are connected by one Mellanox 100 Gbps InfiniBand switch. One machine runs the JVM process while the other provides remote memory via RDMA. All our experiments used a 32GB heap and 4K pages.

Although our application heap size is relatively small (compared to the size of main memory on our machines), the performance of a remote-memory application depends on how much of its working set can fit into local memory and how many (application and GC) threads are used, *not* on how large local memory is. In particular, MemLiner's key data structure is a per-thread PQ (*i.e.*, TQ is not key to MemLiner as it is GC's original data structure). PQ's size depends on the ratio between the number of applications and the number of tracing threads. For instance, for G1, we follow Oracle's recommendation [56] by setting the number of parallel GC threads to be $5 \times$ (core number)/8, and the number of concurrent tracing threads to be 1/4 of the parallel GC threads. With this ratio and a per-thread PQ of 1024 entries, we rarely saw overwrites in our experiments (with our filtering optimizations stated above). However large the heap is, as long as this ratio remains the same, the size of PQ does not need to change; so does the work done by MemLiner.

[2]Its original implementation was incompatible with OpenJDK12.

Spark [74]	**Dataset**	**Size**
MLlib KMeans (**SKM**)	Wikipedia France [4]	1.1GB
Spark Linear Regression (**SLR**)	Wikipedia English [4]	3GB
Spark Transitive Closure (**STC**)	Synthetic graph	1.5M edges 384K vertices
Cassandra [12]	**Workload**	**Operation**
Update Intensive (**CUI**)	Update 50% Insert 50%	10M ops
Read Intensive (**CRI**)	Read 50% Insert 50%	10M ops
Insert Intensive (**CII**)	Insert 50% Update 25% Read 25%	10M ops
Neo4j [52]	**Dataset**	**Size**
PageRank (**NPR**)	Wikipedia Turkish [4]	14M edges 544K vertices
Triangle Counting (**NTR**)	Wikipedia Turkish [4]	14M edges 544K vertices
Degree Centrality (**NDC**)	Dogster Friends [4]	8.5M edges 451K vertices
QuickCached [3]	**Workload**	**Operation**
Write Dominant (**QWD**)	Insert 60% Read 40%	9M ops
Read Dominant (**QRD**)	Insert 20% Read 80%	9M ops
DayTrader [34]	**Workload**	**Size**
Tradesoap (**DTS**)	Synthetic set of stocks	12288 users 8192 sessions

Table 1: Applications and datesets used for G1.

Applications. To evaluate MemLiner, we used a range of cloud applications including Apache Spark [74] (3.0.0), the de-facto data analytics system, Apache Cassandra [12] (3.11), a widely used distributed database, Neo4j [52] (4.3.2), a graph database, QuickCached [3], a Java implementation of Memcached, as well as DayTrader [34], IBM's open-source application emulating an online stock trading system. These applications cover a wide spectrum of text and graph analytics, web services, machine learning tasks, and database query tasks. For each application, their workloads and datasets are reported in Table 1.

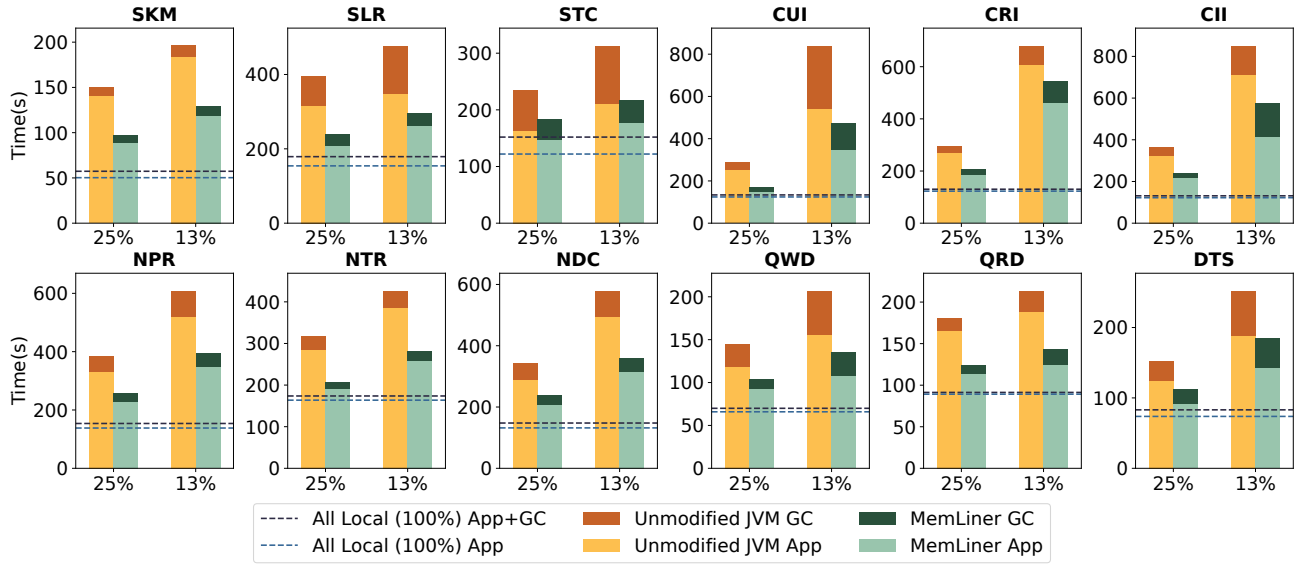


Figure 6: Performance comparisons between G1 GC (yellow bars) and MemLiner (green bars) under two local memory ratios: 25% and 13%; each bar is split into application (bottom with light colors) and GC (top with dark colors) time in seconds. The two dashed lines show application time and total time with unmodified JVM and 100% local memory (no swaps).

The memory access patterns of our applications can be categorized into three types:

- *Mostly sequential access patterns*: Spark applications operate over RDDs. An RDD is an object array or serialized primitive array. Each application thread exhibits clear memory access patterns, *e.g.*, streaming or stride.
- *Random access patterns*: QuickCached (a key-value store) and DayTrader (stock trading simulation) exhibit quite random memory access patterns.
- *Mixed access patterns*: Take Cassandra as an example. Each read/update operation goes through several micro-operations. Different micro-operations have different memory access patterns, *i.e.*, the MemTable loading exhibits a good streaming memory access pattern and some other calculations access memory randomly. Both Cassandra and Neo4j belong to this category.

Our experiments considered two local memory ratios: 25%, and 13% of the total Java heap size (32GB), which are consistent with local memory ratios used in prior work [58, 68]. We enforced these ratios with `cgroup`.

7.2 Performance with G1 GC

Overall. Figure 6 compares the performance of the baseline (the default G1 GC) and MemLiner under two different local memory ratios: 25%, and 13%. As shown, MemLiner offers better performance over the baseline JVM for all workloads, **1.48**× speedup on average under 25% local memory and **1.51**× speedup on average under 13% local memory. A summary of these performance improvements (for the application, GC, and end-to-end performance) is reported in Table 2.

Local Memory	G1 GC			Shenandoah GC		
Configuration	**App**	**GC**	**All**	**App**	**GC**	**All**
25% Local	**1.45**×	**1.65**×	**1.48**×	**1.88**×	**15.33**×	**2.16**×
13% Local	**1.46**×	**1.79**×	**1.51**×	**1.60**×	**6.20**×	**1.80**×

Table 2: Speedups provided by MemLiner for G1 and Shenandoah. (speedup: the average time under each configuration using the unmodified JVM divided by that using MemLiner)

We also compared the number of swap-in pages between MemLiner and the unmodified JVM: MemLiner reduces an average of **81%** of on-demand swap-ins and **56%** of total swap-ins (including both on-demand and prefetching swaps).

Compared with running the whole application in local memory with no swapping (illustrated by dashed lines in Figure 6), the unmodified JVM incurs 2.17× and 3.73× slowdowns under the 25% and 13% local memory configurations, respectively. MemLiner brings them down to 1.47× and 2.48×.

Details. For several workloads (e.g., **SLR**, **STC**, **CUI**, **NDC**, **QWD** and **DTS**), the default JVM's GC time increases dramatically when the local memory ratio drops from 25% to 13%. This is because when memory resources are tight, concurrent tracing becomes slow with many local-memory cache misses. It sometimes cannot finish a complete closure before the heap is full, causing the JVM to pause all application threads and run a time-consuming full-heap GC. Fortunately, MemLiner brings down that GC cost, enabling concurrent

tracing to quickly compute the closure by following the applications' accesses and reducing full-heap GCs.

Cassandra's performance degrades drastically under 13% local memory. In addition to more frequent full-heap GCs, this also stems from data spilling. When the memory usage exceeds a certain ratio (*e.g.*, 2/3) of the heap size, Cassandra automatically spills data from memory to disk. Since concurrent tracing under a tighter local-memory budget becomes much slower, the memory consumption frequently exceeds that ratio, triggering spilling and slowing down the application. In these large-scale systems, GC can actually impact the performance of applications in many unexpected ways.

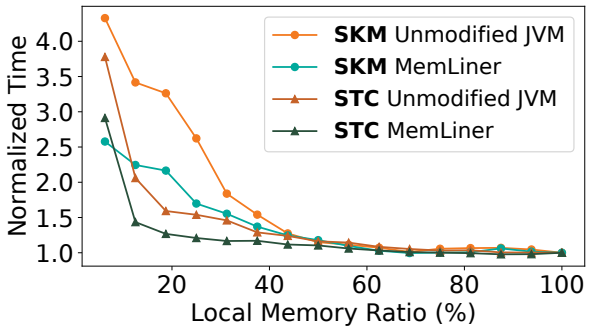


Figure 7: Performance comparisons for **SKM** and **STC** between the unmodified JVM and MemLiner under different local memory configurations.

Different Local Memory Configurations. We ran **SKM** and **STC** with various local-memory ratio configurations and report the performance in Figure 7. As shown, the lower the ratio, the higher the benefit MemLiner provides. For both applications, the turning point is around 50%—MemLiner and the baseline have about the same performance when the local memory ratio reaches 50% or above.

7.3 Performance with Shenandoah GC

To demonstrate the generality of MemLiner, we implemented MemLiner in a second garbage collector: Shenandoah[25], a widely-used highly-concurrent low-pause GC developed by Red Hat. It performs not only concurrent tracing but also concurrent object evaluation to minimize pauses.

Shenandoah provides great latency benefits under sufficient local memory. However, it has extremely poor performance with remote memory involved. For example, the slowdowns under 25% memory for our Spark and Neo4j applications are constantly above 10× and 4×, respectively. Compared to Neo4j, Spark applications usually have much larger working sets, leading to more remote accesses. Such a large overhead highlights the problem of running many concurrent GC threads that do not align with the application's memory access. In particular, Shenandoah is *not* a generational GC (while G1 is). In G1, when the young generation, which contains short-lived objects, is full, the JVM suspends application threads

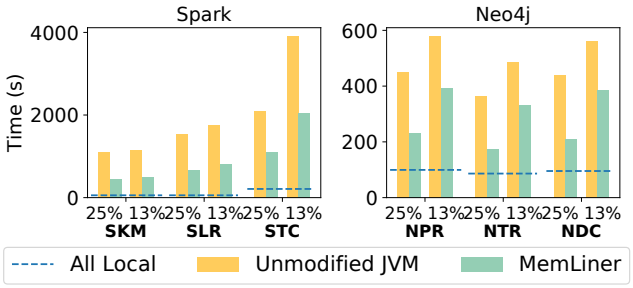


Figure 8: Performance comparison with Shenandoah GC [25].

Spark Programs	Dataset	Size
MLlib KMeans (**SKM**)	Wikipedia Polish [4]	1GB
Spark Linear Regression (**SLR**)	Wikipedia Polish [4]	1GB
Spark Transitive Closure (**STC**)	Synthetic Graph	1.5M edges 384K vertices
Neo4J Programs	**Dataset**	**Size**
PageRank (**NPR**)	Wikipedia Slovak [4]	7.6M edges 291K vertices
Triangle Counting (**NTR**)	Wikipedia Slovak [4]	7.6M edges 291K vertices
Degree Centrality (**NDC**)	Wikipedia min-nan [4]	4.4M edges 429K vertices

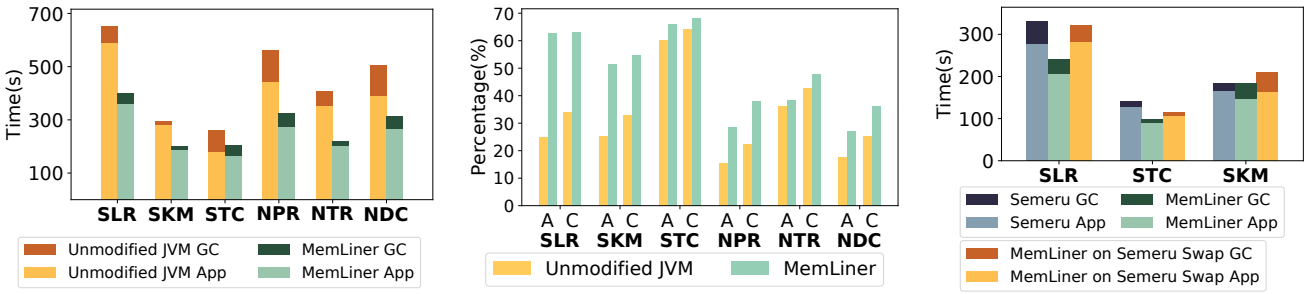
Table 3: Benchmarks and datasets for Shenandoah.

and evacuates objects in the young generation. This leads to excellent data locality after evacuation. However, under Shenandoah GC, the JVM runs concurrent tracing much more frequently to scan the full heap to identify and collect garbage. Those tracing threads exhibit particularly poor locality. To evaluate Shenandoah, we had to use smaller datasets (Table 3) for a tolerable running time.

As illustrated in Figure 8 and summarized in Table 2, MemLiner achieves an overall **2.16**× and **1.80**× speedup compared to the unmodified JVM under 25% and 13% local memory, respectively. MemLiner reduces an average of 82% on-demand swap-ins and 56% of total swap-ins under 25% local memory, while it reduces 79% of on-demand swap-ins and 22% of total swap-ins under 13% local memory. As shown in Table 2, MemLiner provides tremendous improvements for Shenandoah's GC performance, because the unmodified JVM frequently triggers *full-heap stop-the-world GC*.

7.4 Comparisons with Other Systems

Leap [48] is an advanced OS-level prefetcher. It uses a majority-vote algorithm to determine how to do prefetches. In cases where no clear access patterns are seen, Leap aggressively prefetches consecutive pages. Although this strategy may improve performance for native applications whose memory accesses often fall into large arrays, it often hurts managed applications such as Spark, as GC's pointer-chasing behavior often makes prefetched consecutive pages useless.



(a) Leap: Running time (b) Leap: Accuracy(A) and Coverage(C) (c) Semeru: Running time

Figure 9: Performance comparisons with Leap and Semeru; Semeru crashed on **NPR**, **NTR**, and **NDC** (*i.e.*, Neo4j applications).

Our hypothesis is that even aggressive prefetchers like Leap cannot handle the interference of GC, and that by aligning the memory accesses of GC with application threads, MemLiner can improve application performance under Leap just like under less aggressive prefetchers. To test our hypothesis, we compared MemLiner with the unmodified JVM (default G1 GC) both using Leap as the prefetcher. This experiment was conducted on three Spark applications: **SLR**, **SKM**, **STC**, and three Neo4j applications: **NPR**, **NTR**, **NDC**, under 25% local memory.

As shown in Figure 9(a), compared with the unmodified JVM on Leap, MemLiner improves the overall performance by an average of 1.6× and reduces 58% of on-demand swap-ins, as well as 53% of total swap-ins on average. To understand whether MemLiner improves Leap's prefetching effectiveness, we additionally measured Leap's prefetching *accuracy* (i.e., the percentage of page faults hitting on the swap cache among prefetched pages) and *coverage* (i.e., the percentage of swap cache hits among all page faults) with and without MemLiner. As shown in Figure 9(b), MemLiner helps Leap deliver higher accuracy and coverage. We still observed that MemLiner is not as useful for **STC** and **NTR** as it is for the two applications. This is because the number of live objects in **STC** during concurrent tracing is relatively small, leading to shorter tracing time and better access patterns. For **NTR**, its application threads exhibit random memory accesses themselves. Hence, Leap cannot detect clear patterns even if MemLiner has already eliminated much of the interference.

Semeru [68] is a memory-disaggregated runtime, where the entire Java heap is backed by physical memory on memory servers and the CPU server's local memory is used as an inclusive cache. Semeru completely redesigned the JVM so that all the garbage collection is offloaded from the CPU server to the memory servers, through special lightweight JVMs running there. Applications execute on the CPU server with absolutely no GC interference, at the cost of extra computation on memory servers (i.e., two extra cores for each memory server to run the offloaded lightweight JVM).

Here, to evaluate whether MemLiner can achieve similar performance as Semeru, without Semeru's intrusive changes to JVM and Semeru's extra computation load on memory servers, we ran the same three Spark applications under 25% local memory on top of (1) Semeru, (2) MemLiner on Semeru's swap system (*i.e.*, a modified version of NVMe-over-fabrics [1]), and (3) MemLiner on FastSwap [10], which is the default swap system MemLiner builds on. We ran Semeru with one CPU server and two memory servers—the Java heap is partitioned between the memory servers.

As shown in Figure 9(c) , MemLiner's performance is comparable with Semeru when using Semeru's swap system, and is much better than Semeru when using MemLiner's default swap system. The reason is that, even though Semeru completely eliminates GC tracing threads from the local machine, it has to perform a great deal of coordination between servers to handle cross-server references, incurring communication overheads. We would have also liked to run Semeru directly over FastSwap, but this was not feasible due to Semeru's runtime-kernel co-design that prevents Semeru from easily adapting to different swap systems.

We could not directly compare Memliner with AIFM [58] as AIFM targets native languages (C/C++) applications and requires rewriting programs. However, the major idea behind AIFM—swapping at the object granularity—is orthogonal to MemLiner. MemLiner can also benefit from a redesigned swap system that performs object-level swapping.

7.5 More Detailed Results

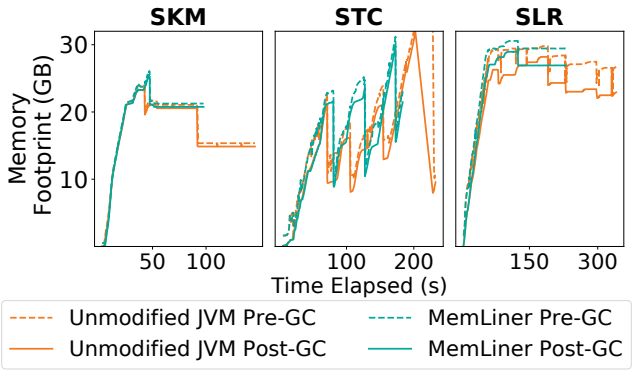


Figure 10: Memory footprints for **SKM**, **STC**, and **SLR**, between unmodified JVM and MemLiner under 25% rate.

Memory Reclamation Impact. Since MemLiner postpones tracing objects estimated to be remote, it may delay memory reclamation. To understand the impact of such a delay, we collected post-GC memory footprints for **STC**, **SKM**, and **SLR** executed atop the unmodified JVM and MemLiner under 25% local memory configuration. Figure 10 reports, for each program, both its pre-GC and post-GC memory footprints. As shown, for all three workloads, MemLiner incurs insignificant delays in memory reclamation and only a slight increase in the peak memory consumption. This is because tracing of each remote object can only be postponed a few times (*i.e.*, *MAX_DL*); when the available heap runs low, *MAX_DL* becomes 0 and we do not postpone GC at all.

Epoch Estimation Effectiveness. We collected the number of objects that are scanned from PQ and TQ for three Spark applications under 25% local memory. The ratio of objects scanned from PQ over total objects scanned during the concurrent tracing phase is 45%, 42%, and 11% respectively for **SLR**, **SKM** and **STC**. We also evaluated MemLiner after disabling epoch estimation: we saw an overall performance degradation of 8.6%, 8.8% and 11.3% respectively, for **SLR**, **SKM** and **STC** under 25% local memory.

8 Related Work

Far Memory. Due to rapid technological advances in network controllers, it has become practical to reorganize resources into disaggregated clusters [32, 18, 27, 15]. A disaggregated cluster can increase the hardware resource utilization and has the potential to overcome fundamental hardware limits, such as the critical "memory capacity wall" [14, 44, 43, 67, 20, 38, 7, 11]. A body of techniques [10, 30, 6, 58, 68, 61, 63, 68, 31, 69] have been developed to enable applications to use remote memory and efficiently access remote data.

Among these techniques, a mainstream approach [10, 30, 6] is to provide *transparent* remote memory access with *swap mechanisms* where the running application is not aware of remote memory, which is mapped into the application host server as a *swap partition*. The host server reserves a certain amount of local memory as a software-managed data cache. Once the program accesses a page that does not reside in the data cache, it triggers a page fault, and the swap system fetches the page from a remote memory server via RDMA.

A traditional swap system was designed for *slow and rare* accesses to disks, *not for fast and frequent* accesses to remote memory via RDMA. Having realized this speed discrepancy, existing techniques have performed a variety of optimizations, *e.g.*, removing redundant block layers [30], leveraging multi-queues [10], or performing per-application prefetching [48], all to maximize the paging/swap efficiency. Despite these commendable efforts, these techniques need to pay a "transparency tax"—since all remote accesses go through the OS kernel, which incurs a non-trivial overhead. To mitigate such a software-introduced overhead, work such as AIFM [58] provides primitives for developers to perform efficient remote access in the user space. AIFM outperforms swap-based techniques by bypassing the kernel data plane. However, to use AIFM, applications have to be rewritten (with new primitives), which can significantly hinder its practical use.

Modern Garbage Collectors. Modern GCs, including Oracle's Garbage-First (G1) GC [22], Red Hat's Shenandoah GC [25], Azul's pauseless GC [21], and C4 [64], all use concurrent tracing. Some also perform concurrent memory compaction [39, 2]. As big data systems gain popularity, there is a line of work that develops systems for applications running on the cloud [24, 53, 54, 51, 67], on NUMA machines [29], as well as using non-volatile memory [67, 71, 9]. Yak [53] is a region-based big-data-friendly GC. Taurus [47] coordinates GC efforts among workers in a distributed system. Facade [54] uses region-based memory management to reduce GC costs for Big Data applications. Gerenuk [51] develops a compiler analysis and runtime system that enable native representation of data for managed analytics systems such as Spark and Hadoop. Espresso [71] and Panthera [67] are designed for systems with non-volatile memory. Platinum [70] aims to reduce tail latency for interactive applications. NUMAGiC [29] is a GC that provides efficiency by considering NUMA features.

Semeru [68] and Mako [46] are both GCs developed for memory disaggregation. While they both achieve superior performance via compute offloading (*e.g.*, running concurrent tracing and evacuation on memory servers), offloading introduces numerous challenges in resource utilization and cluster scheduling. AIFM [58] performs GC-like memory compaction to eliminate dead objects to reduce read/write amplification. This approach is orthogonal to MemLiner, which leverages tracing for prefetching.

9 Conclusion

This paper presents MemLiner, a runtime technique that reduces the GC-application interference by aligning the memory accesses of application and tracing threads. We classify reachable objects into three categories and treat objects in each category in a different way to achieve the two seemingly conflicting goals. Our promising results with two production GCs demonstrate that MemLiner can be readily used in today's datacenters.

Acknowledgments

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Aurojit Panda for his feedback. This work is supported by NSF grants CNS-1703598, CNS-1763172, CNS-1764039, CNS-1907352, CNS-1956180, CNS-2007737, CNS-2006437, CNS-2128653, CNS-2106838, CCF-2119184, ONR grant N00014-18-1-2037, and research grants from Facebook, Microsoft, and Cisco.

A Artifact Appendix

A.1 Artifact Summary

MemLiner is a managed runtime built for a memory-disaggregated cluster where each managed application runs on one server and uses both local memory and remote memory located on another server. When launched on MemLiner, the process fetches data from the remote server via the paging system. MemLiner reduces the local-memory working set and improves the remote-memory prefetching by lining up the memory accesses from application and GC threads. MemLiner is transparent to applications and can be integrated in any existing GC algorithms, such as G1 and Shenandoah.

A.2 Artifact Check-list

- **Hardware: Intel servers with InfiniBand**
- **Run-time environment: OpenJDK 12.02, Linux-5.4, Ubuntu 18.04 with MLNX-OFED 4.9-2.2.4.0**
- **Public link: https://github.com/uclasystem/MemLiner**
- **Code licenses: The GNU General Public License (GPL)**

A.3 Description

A.3.1 MemLiner's Codebase

MemLiner contains the following three components:

- the Linux kernel, which includes a modified swap system,
- the Java Virtual Machine (JVM) with MemLiner,
- necessary shell scripts and configuration files.

A.3.2 Deploying MemLiner

To build MemLiner, the first step is to download its source code:

```
git clone
git@github.com:uclasystem/MemLiner.git
```

When deploying MemLiner, install the components in the following order: (1) install the kernel and the RDMA module on all participating servers; (2) install the JVM with MemLiner on the server that runs the process; (3) connect the participating servers before running applications.

Kernel Installation. We first discuss how to build and install the kernel.

- Modify `grub` and set `transparent_hugepage` to `madvise`:

```
sudo vim /etc/default/grub
+ transparent_hugepage=madvise
```

- Install the kernel and restart the machine:

```
cd MemLiner/Kernel
sudo ./build_kernel.sh build
sudo ./build_kernel.sh install
```

- Install the MLNX OFED driver:

  MemLiner has only been tested on Ubuntu 18.04 with MLNX-OFED-4.9-2.2.4.0. The driver should be installed all participating servers.

```
# @all participating servers
# Remove the incompatible libraries
sudo apt remove ibverbs-providers:amd64
librdmacm1:amd64 librdmacm-dev:amd64
libibverbs-dev:amd64 libopensm5a
libosmvendor4 libosmcomp3 -y

# Download and install the MLNX OFED driver
curl https://content.mellanox.com/ofed/
MLNX_OFED-4.9-2.2.4.0/MLNX_OFED_LINUX
-4.9-2.2.4.0-ubuntu18.04-x86_64.tgz
--output MLNX_OFED.tgz
tar -xzf MLNX_OFED.tgz
sudo MLNX_OFED/mlnxofedinstall
--add-kernel-support

# Enable the openidb and opensmd services
sudo systemctl enable openibd
sudo systemctl start openibd
sudo systemctl enable opensmd
sudo systemctl start opensmd
```

- Configure and install the MemLiner RDMA module:

```
# Assign the IP of a memory server into:
# @CPU server
# MemLiner/rswap/client/rswap_rdma.c
char ip[] = "10.0.0.4"; # IP of memory server
# @memory server
# MemLiner/rswap/server/rswap_server.cpp
const char *ip_str = "10.0.0.4";

# Build the MemLiner RDMA module
# @CPU server
cd MemLiner/rswap/client
make clean && make
# @memory server
cd MemLiner/rswap/server
make clean && make
```

Install the MemLiner (JVM). We next discuss the steps to build and install the MemLiner JVM on the CPU server.

- Download Oracle JDK 12 to build the MemLiner JVM:

```
# @CPU server
# Assume jdk 12.02 is under path:
# ${HOME}/jdk-12.0.2
cd MemLiner/JDK
./configure -with-boot-jdk=${HOME}/jdk-
12.0.2 --with-debug-level=release
make JOBS=32

# Run the applications with the built JVM.
# The built JVM (MemLiner) is under:
MemLiner/JDK/build/
linux-x86_64-server-release/jdk
```

A.3.3 Running Applications

To run applications, we first need to connect the CPU and memory servers. Next, we mount the remote memory pool as a swap partition on the CPU server. When the application uses more memory than the limit set by `cgroup`, its data will be swapped out to remote memory via RDMA.

- Launch memory servers:

```
# @memory server
cd MemLiner/rswap/server
./rswap-server
```

- Connect the CPU server with memory servers:

```
# @CPU server
cd MemLiner/rswap/client
./manage_rswap_client.sh install
```

- Set a cache size limit for an application:

```
# For example, create a cgroup with a 9GB
memory limit.
# @CPU server
# Create the cgroup with the name memctl
# $USER is the username of the account
sudo cgcreate -t $USER -a $USER -g
memory:/memctl

# Set the memory limit to 9GB
echo 9g > /sys/fs/cgroup/memory/
memctl/memory.limit_in_bytes
```

- Add a Spark executor into `cgroup`:

```
# Add a Spark worker into cgroup, memctl.
# Its sub-process, executor, falls into the same
cgroup.
# @CPU server
# Modify the function start_instance under:
# Spark/sbin/start-slave.sh
cgexec -sticky -g memory:memctl
"${SPARK_HOME}/sbin" /sparkdaemon.sh
start $CLASS $WORKER_NUM -webui-port
"$WEBUI_PORT" $PORT_FLAG $PORT_NUM
$MASTER "$@"
```

- Launch the Spark cluster:

  Certain JVM options need to be added to run the MemLiner. We use the Spark as an example here. Please refer to the MemLiner's code repository for more details about how to run other applications.

```
# @CPU server
# Replace the Spark default configuration
cd ${spark-home-dir}/conf
cp MemLiner/config-files/spark-confs/
spark-defaults-memliner.conf
spark-defaults.conf

# Launch the Spark master and worker services
${spark-home-dir}/sbin/start-all.sh
```

- Run Spark applications:

  Specify the Spark application name and local memory ratio, *e.g.*, 25% or 13%, and then execute the applications:

```
# @CPU server
# Para#1 application: lr, km, tc
# Para#2 mem_local_ratio: 25, 13
MemLiner/app-scripts/memliner.sh
${application} ${mem_local_ratio}
```

  More details of MemLiner's installation and deployment can be found in MemLiner's code repository.

References

[1] NVMe over fabrics. http://community.mellanox.com/s/article/what-is-nvme-over-fabrics-x.

[2] The Z garbage collector. https://wiki.openjdk.java.net/display/zgc/Main.

[3] QuickCached. https://github.com/QuickServerLab/QuickCached, 2017.

[4] Konect network datasets. http://konect.cc/networks/, 2021.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[6] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SoCC*, pages 121–127, 2017.

[7] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal. Designing far memory data structures: Think outside the box. In *HotOS*, pages 120–126, 2019.

[8] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, pages 336–348, 2015.

[9] S. Akram, J. B. Sartor, S. M. Blackburn, K. S. McKinley, and L. Eeckhout. Write-rationing garbage collection for hybrid memories. In *PLDI*, pages 62–77, 2018.

[10] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[11] S. Angel, M. Nanavati, and S. Sen. Disaggregation and the application. In *HotCloud*, 2020.

[12] Apache. Apache cassandra. https://cassandra.apache.org, 2021.

[13] K. Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers. In *FAST*, 2014.

[14] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[15] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA*, 2011.

[16] M. N. Bojnordi and E. Ipek. PARDIS: A programmable memory controller for the DDRx interfacing standards. In *ISCA*, pages 13–24, 2012.

[17] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

[18] A. Carbonari and I. Beschasnikh. Tolerating faults in disaggregated datacenters. In *HotNets-XVI*, pages 164–170, 2017.

[19] CCIX. Cache coherent interconnect for accelerators. https://www.ccixconsortium.com/, 2018.

[20] L. Chen, J. Zhao, C. Wang, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, G. H. Xu, and H. Cui. Unified holistic memory management supporting multiple big data processing frameworks over hybrid memories. *ACM Trans. Comput. Syst.*, 2022.

[21] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *VEE*, pages 46–56, 2005.

[22] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM*, pages 37–48, 2004.

[23] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[24] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP*, pages 394–409, 2015.

[25] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *PPPJ*, pages 13:1–13:9, 2016.

[26] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *OSDI*, pages 281–297, 2020.

[27] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

[28] GenZ. Genz consortium. http://genzconsortium.org/, 2019.

[29] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. NumaGiC: A garbage collector for big data on big NUMA machines. In *ASPLOS*, pages 661–673, 2015.

[30] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[31] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *ASPLOS*, pages 417–433, 2022.

[32] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

[33] Hewlett-Packard. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/, 2015.

[34] IBM. Daytrader. https://www.ibm.com/docs/en/linux-on-systems?topic=bad-daytrader, 2021.

[35] Intel. Intel high performance computing fabrics. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/, 2019.

[36] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[37] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, pages 295–306, 2014.

[38] K. Keeton. The Machine: An architecture for memory-centric computing. In *ROSS*, 2015.

[39] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental, and parallel compaction. In *PLDI*, pages 354–363, 2006.

[40] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, pages 317–330, 2019.

[41] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA*, pages 2–13, 2009.

[42] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, pages 429–444, 2014.

[43] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, pages 267–278, 2009.

[44] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, pages 1–12, 2012.

[45] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Big Data*, pages 2884 – 2892, 2017.

[46] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, pages 92–107, 2022.

[47] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS*, pages 457–471, 2016.

[48] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX ATC*, pages 843–857, 2020.

[49] Mellanox. Connectx-6 single/dual-port adapter supporting 200gb/s with vpi. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card, 2019.

[50] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2), 2016.

[51] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu. Gerenuk: Thin computation over big native data using speculative program transformation. In *SOSP*, pages 538–553, 2019.

[52] Neo4j. Neo4j graph data platform. https://neo4j.com, 2021.

[53] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365, 2016.

[54] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, pages 675–690, 2015.

[55] OpenCAPI. Open coherent accelerator processor interface. https://opencapi.org/, 2018.

[56] Oracle. Garbage first garbage collector tuning. https://www.oracle.com/technical-resources/articles/java/g1gc.html, 2020.

[57] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[58] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*, pages 315–332, 2020.

[59] S. M. Rumble. Infiniband verbs performance. https://ramcloud.atlassian.net/wiki/display/RAM/Infiniband+Verbs+Performance, 2010.

[60] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable SSD. In *OSDI*, pages 67–80, 2014.

[61] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[62] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, pages 255–270, 2019.

[63] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart remote memory. In *EuroSys*, 2020.

[64] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In *ISMM*, pages 79–88, 2011.

[65] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *EuroSys*, 2020.

[66] M. Tork, L. Maudlej, and M. Silberstein. Lynx: A SmartNIC-driven accelerator-centric architecture for network servers. In *ASPLOS*, pages 117–131, 2020.

[67] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *PLDI*, pages 347–362, 2019.

[68] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *OSDI*, pages 261–280, 2020.

[69] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. https://arxiv.org/abs/2203.09615, 2022.

[70] M. Wu, Z. Zhao, Y. Yang, H. Li, H. Chen, B. Zang, H. Guan, S. Li, C. Lu, and T. Zhang. Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services. In *USENIX ATC*, pages 159–172, 2020.

[71] M. Wu, Z. Ziming, L. Haoyu, L. Heting, C. Haibo, Z. binyu, and G. Haibing. Espresso: Brewing Java for more non-volatility. In *ASPLOS*, pages 70–83, 2018.

[72] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[73] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

[74] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

Carbink: Fault-Tolerant Far Memory

Yang Zhou[†*] Hassan M.G. Wassel[‡] Sihang Liu[§*] Jiaqi Gao[†] James Mickens[†] Minlan Yu[†‡]
Chris Kennelly[‡] Paul Turner[‡] David E. Culler[‡] Henry M. Levy[‖‡] Amin Vahdat[‡]

[†]*Harvard University* [‡]*Google* [§]*University of Virginia* [‖]*University of Washington*

Abstract

Far memory systems allow an application to transparently access local memory as well as memory belonging to remote machines. Fault tolerance is a critical property of any practical approach for far memory, since machine failures (both planned and unplanned) are endemic in datacenters. However, designing a fault tolerance scheme that is efficient with respect to both computation and storage is difficult. In this paper, we introduce Carbink, a far memory system that uses erasure-coding, remote memory compaction, one-sided RMAs, and offloadable parity calculations to achieve fast, storage-efficient fault tolerance. Compared to Hydra, a state-of-the-art fault-tolerant system for far memory, Carbink has 29% lower tail latency and 48% higher application performance, with at most 35% higher memory usage.

1 Introduction

In a datacenter, matching a particular application to just enough memory and CPUs is hard. A commodity server tightly couples memory and compute, hosting a fixed number of CPUs and RAM modules that are unlikely to exactly match the computational requirements of any particular application. Even if a datacenter contains a heterogeneous mix of server configurations, the load on each server (and thus the amount of available resources for a new application) changes dynamically as old applications exit and new applications arrive. Thus, even state-of-the-art cluster schedulers [51, 52] struggle to efficiently bin-pack a datacenter's aggregate collection of CPUs and RAM. For example, Google [52] and Alibaba [34] report that the average server has only ~60% memory utilization, with substantial variance across machines.

Memory is a particularly vexing resource for two reasons. First, for several important types of applications [19, 20, 33, 54], the data set is too big to fit into the RAM of a single machine, even if the entire machine is assigned to a single application instance. Second, for these kinds of applications, alleviating memory pressure by swapping data between RAM and storage [14] would lead to significant application slowdowns, because even SSD accesses are orders of magnitude slower than RAM accesses. For example, Google runs a graph analysis engine [28] whose data set is dozens of GBs in size. This workload runs 46% faster when it shuffles data purely through RAM instead of between RAM and SSDs.

Disaggregated datacenter memory [2, 5, 15, 16, 22, 44, 46] is a promising solution. In this approach, a CPU can be paired with an arbitrary set of possibly-remote RAM modules, with a fast network interconnect keeping access latencies to far memory small. From a developer's perspective, far memory can be exposed to applications in several ways. For example, an OS can treat far RAM as a swap device, transparently exchanging pages between local RAM and far RAM [5, 22, 46]. Alternatively, an application-level runtime like AIFM [44] can expose remotable pointer abstractions to developers, such that pointer dereferences (or the runtime's detection of high memory pressure) trigger swaps into and out of far memory.

Much of the prior work on disaggregated memory [2, 44, 55] has a common limitation: a lack of fault tolerance. Unfortunately, in a datacenter containing hundreds of thousands of machines, faults are pervasive. Many of these faults are planned, like the distribution of kernel upgrades that require server reboots, or the intentional termination of a low-priority task when a higher-priority task arrives. However, many server faults are unpredictable, like those caused by hardware failures or kernel panics. Thus, any *practical* system for far memory has to provide a scalable, fast mechanism to recover from unexpected server failures. Otherwise, the failure rate of an application using far memory will be much higher than the failure rate of an application that only uses local memory; the reason is that the use of far memory increases the set of machines whose failure can impact an application [8].

Some prior far-memory systems do provide fault tolerance via replication [5, 22, 46]. However, replication-based approaches suffer from high storage overheads. Hydra [29] uses erasure coding, which has smaller storage penalties than replication. However, Hydra's coding scheme stripes a single memory page across multiple remote nodes. This means that a compute node requires multiple network fetches to reconstruct a page; furthermore, computation over that page cannot be outsourced to remote memory nodes, since each node contains only a subset of the page's bytes.

[*]Contributed to this work during internships at Google.

In this paper, we present Carbink,[1] a new framework for far memory that provides efficient, high-performance fault recovery. Like (non-fault-tolerant) AIFM, Carbink exposes far memory to developers via application-level remoteable pointers. When Carbink's runtime must evict data from local RAM, Carbink writes erasure-coded versions of that data to remote memory nodes. The advantage of erasure coding is that it provides equivalent redundancy to pure replication, while avoiding the double or triple storage overheads that replication incurs. However, straightforward erasure coding is a poor fit for the memory data created by applications written in standard programming languages like C++ and Go; those applications allocate variable-sized memory objects, but erasure coding requires equal-sized blocks. To solve this problem, Carbink eschews the object-granularity swapping strategy of AIFM, and instead swaps at the granularity of *spans*. A single span consists of multiple memory pages that contain objects with similar sizes. Carbink's runtime asynchronously and transparently moves local objects within the spans in local memory, grouping cold objects together and hot objects together. When necessary, Carbink batch-evicts cold spans, calculating parity bits for those spans at eviction time, and writing the associated fragments to remote memory nodes. Carbink utilizes one-sided remote memory accesses (RMAs) to efficiently perform swapping activity, minimizing network utilization. Unlike Hydra, Carbink's erasure coding scheme allows a compute node to fetch a far memory region using a single network request.

In Carbink, each span lives in exactly one place: the local RAM of a compute node, or the far RAM of a memory node. Thus, swapping a span from far RAM to local RAM creates dead space (and thus fragmentation) in far RAM. Carbink runs pauseless defragmentation threads in the background, asynchronously reclaiming space to use for later swap-outs.

We have implemented Carbink atop our datacenter infrastructure. Compared to Hydra, Carbink has up to 29% lower tail latency and 48% higher application performance, with at most 35% more remote memory usage. Unlike Hydra, Carbink also allows computation to be offloaded to remote memory nodes.

In summary, this paper has four contributions:

- a span-based approach for solving the size mismatch between the granularity of erasure coding and the size of the objects allocated by compute nodes;
- new algorithms for defragmenting the RAM belonging to remote memory nodes that store erasure-encoded spans;
- an application runtime that hides spans, object migration within spans, and erasure coding from application-level developers; and
- a thorough evaluation of the performance trade-offs made by different approaches for adding fault tolerance to far memory systems.

[1] Carbink is a Pokémon that has a high defense score.

2 Background

Recent work on far memory has used one of two approaches. The first approach modifies the OS that runs applications, exploiting the fact that preexisting OS abstractions already decouple application-visible in-memory data from the backing storage hierarchy. For example, INFINISWAP [22], Fastswap [5], and LegoOS [46] leverage virtual memory support to swap application memory to far RAM instead of a local SSD or hard disk. Applications use standard language-level pointers to interact with memory objects; behind the scenes, the OS swaps pages between local RAM and far RAM, e.g., in response to page faults for non-locally-resident pages. In contrast, the remote region approach [2] exposes far memory via file system abstractions. Applications name remote memory regions using standard filenames, and interact with regions using standard file operations like `open()` and `read()`.

Exposing far memory via OS abstractions is attractive because it requires minimal changes to application-level code. However, invasive kernel changes are needed; such changes require substantial implementation effort, and are difficult to maintain as other parts of the kernel evolve.

The second far-memory approach requires more help from application-level code. For example, AIFM [44] uses a modified C++ runtime to hide the details of managing far memory. The runtime provides special pointer types whose dereferencing may trigger the swapping of a remote C++-level object into local RAM. AIFM's runtime tracks object hotness using GC-style read/write barriers, and uses background threads to swap out cold local objects when local memory pressure is high. To synchronize the local memory accesses generated by application threads and runtime threads, AIFM embeds a variety of metadata bits (e.g., `present`, `isBeingEvicted`) in each smart pointer, leveraging an RCU-like scheme [36] to protect concurrent accesses to a pointer's referenced object.

Listing 1 provides an example of how applications use AIFM's smart pointers. Like AIFM, Carbink exposes far memory via smart pointers, but unlike AIFM, Carbink provides fault tolerance.

3 Carbink Design

Figure 1 depicts the high-level architecture of Carbink. **Compute nodes** execute single-process (but potentially multi-threaded) applications that want to use far memory. **Memory nodes** provide far memory that compute nodes use to store application data that cannot fit in local RAM. A logically-centralized **memory manager** tracks the liveness of compute nodes and memory nodes. The manager also coordinates the assignment of far memory **regions** to compute nodes. When a memory node wants to make a local memory region available to compute nodes, the memory node *registers* the region with the memory manager. Later, when a compute node requires far memory, the compute node sends an *allocation* request to the memory manager, who then assigns a registered, unallo-

```
RemUniquePtr<Node> rem_ptr = AIFM::MakeUnique<Node>();
{
  DerefScope scope;
  Node* normal_ptr = rem_ptr.Deref(scope);
  computeOverNodeObject(normal_ptr);
} // Scope is destroyed; Node object can be evicted.
```

Listing 1: Example of how AIFM applications interact with far memory. In the code above, the application first allocates a `Node` object that is managed by a particular `RemUniquePtr`. Such a remote unique pointer represents a pointer to an object that (1) can be swapped between local and far memory, and (2) can only be pointed to by a single application-level pointer. The code then creates a new scope via an open brace, declares a `DerefScope` variable, and invokes the `RemUniquePtr`'s `Deref()` method, passing the `DerefScope` variable as an argument. `Deref()` essentially grabs an RCU lock on the remotable memory object, and returns a normal C++ pointer to the application. After the application has finished using the normal pointer, the scope terminates and the destructor of the `DerefScope` runs, releasing the RCU lock and allowing the object to be evicted from local memory.

cated region. Upon receiving a *deallocation* message from a compute node, the memory manager marks the associated region as available for use by other compute nodes. A memory node can ask the memory manager to *deregister* a previously registered (but currently unallocated) region, withdrawing the region from the global pool of far memory.

Carbink does not require participating machines to use custom hardware. For example, any machine in a datacenter can be a memory node if that machine runs the Carbink memory host daemon. Similarly, any machine can be a compute node if that node's applications use the Carbink runtime.

From the perspective of an application developer, the Carbink runtime allows a program to dynamically allocate and deallocate memory objects of arbitrary size. As described in Section 3.2, programs access those objects through AIFM-like remotable pointers [44]. When applications dereference pointers that refer to non-local (i.e., swapped-out) objects, Carbink pulls the desired objects from far memory. Under the hood, Carbink's runtime manages objects using **spans** (§3.3) and **spansets** (§3.4). A span is a contiguous run of memory pages; a single region allocated by a compute node contains one or more spans. Similar to slab allocators like Facebook's jemalloc [17] and Google's TCMalloc [21, 24], Carbink rounds up each object allocation to the bin size of the relevant span, and aligns each span to the page size used by compute nodes and memory nodes. Carbink swaps far memory into local memory at the granularity of a span; however, when local memory pressure is high, Carbink swaps local memory out to far memory at the granularity of a spanset (i.e., a collection of spans of the same size). In preparation for

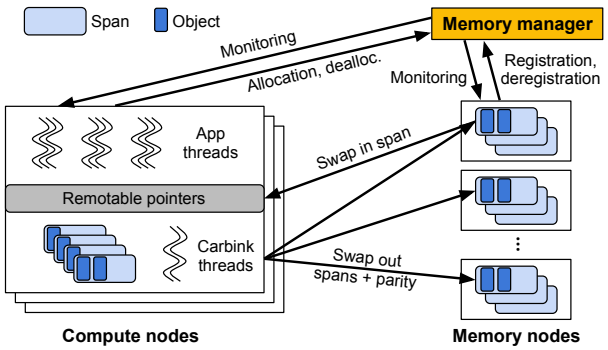


Figure 1: Carbink's high-level architecture.

swap-outs, background threads on compute nodes group cold objects into cold spans, and bundle a group of cold spans into a spanset; at eviction time, the threads generate erasure-coding parity data for the spanset, and then evict the spanset and the parity data to remote nodes. As we discuss in Sections 3.4 and 3.5, this approach simplifies memory management and fault tolerance.

Carbink disallows cross-application memory sharing. This approach is a natural fit for our target applications, and has the advantage of simplifying failure recovery and avoiding the need for expensive coherence traffic [46].

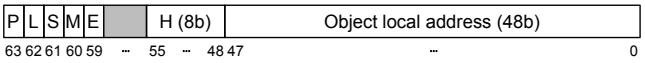
3.1 Failure Model

Carbink implements the logically-centralized memory manager as a replicated state machine [1, 45]. Thus, Carbink assumes that the memory manager will not fail. Carbink assumes that memory nodes and compute nodes may experience fail-stop faults. Carbink does not handle Byzantine failures or partial network failures.

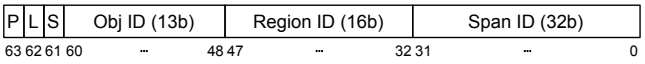
The memory manager tracks the liveness of compute nodes and memory nodes via heartbeats. When a compute node fails, the memory manager instructs the memory nodes to deallocate the relevant spans; if applications desire, they can use an application-level fault tolerance scheme like checkpointing to ensure that application-level data is recoverable. When a memory node fails, the memory manager deregisters the node's regions from the global pool of far memory. However, erasure-coding recovery of the node's regions is initiated by a compute node when the compute node unsuccessfully tries to read or write a span belonging to the failed memory node. If an application thread on a compute node tries to read a span that is currently being recovered, the read will use Carbink's degraded read protocol (§3.5), reconstructing the span using data from other spans and parity blocks.

3.2 Remotable Pointers

Like AIFM, Carbink exposes far memory through C++-level smart pointers. However, as shown in Figure 2, Carbink uses a different pointer encoding to represent span information.



(a) Local object.



(b) Far object.

Field	Meaning
Present	Is the object in local RAM or far RAM?
Lock	Is the object (spin)locked by a thread?
Shared	Is the pointer a unique pointer or a shared pointer?
Moving	Is the object being moved by a background thread?
Evicting	Is the object being evicted by a background thread?
Hotness	Is the object frequently accessed?

(c) Field semantics.

Figure 2: Carbink's `RemUniquePtr` representation. In contrast to AIFM [44], Carbink does not embed information about a data structure ID or an object size. Instead, Carbink embeds span metadata (namely, a Region ID and a Span ID) to associate a pointed-to object with its backing span.

A Carbink `RemUniquePtr` has the same size as a traditional `std::unique_ptr` (i.e., 8 bytes). The **P**resent bit indicates whether the pointed-to object resides in local RAM. The **S**hared bit indicates whether a pointer implements unique-pointer semantics or shared-pointer semantics; the former only allows a single reference to the pointed-to object. The **L**ock, **M**oving, and **E**victing bits are used to synchronize object accesses between application threads and Carbink's background threads (§3.6). The **H**otness byte is consulted by the background threads when deciding whether an object is cold (and thus a priority for eviction).

If an object is local, the local virtual address of the object is directly embedded in the pointer. If an object has been evicted, the pointer describes how to locate the object. In particular, the Obj ID indicates the location of an object within a particular span; the Span ID identifies that span; and the Region ID denotes the far memory region that contains the span.

Carbink supports two smart pointer types: `RemUniquePtr`, which only allows one reference to the underlying object, and `RemSharedPtr`, which allows multiple references. When moving or evicting an object, Carbink's background threads need a way to locate and update the smart pointer(s) which reference the object. To do so, Carbink uses AIFM's approach of embedding a "reverse pointer" in each object; the reverse pointer points to the object's single `RemUniquePtr`, or to the first `RemSharedPtr` that references the object. An individual `RemSharedPtr` is 16 bytes large, with the last 8 bytes storing a pointer that references the next `RemSharedPtr` in the list. Thus, Carbink's runtime can find all of an object's `RemSharedPtrs` by discovering the first one via the object's reverse pointer, and then iterating across the linked list.

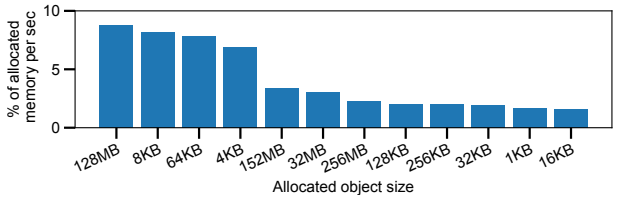


Figure 3: Allocation sizes in our production workloads.

3.3 Span-Based Memory Management

Local memory management: A span is a contiguous set of pages that contain objects of the same size class. Carbink supports 86 different size classes, and aligns each span on an 8KB boundary; Carbink borrows these configuration parameters from TCMalloc [21, 24], which observed these parameters to reduce internal fragmentation. When an application allocates a new object, Carbink tries to round the object size up to the nearest size class and assign a free object slot from an appropriate span. If the object is bigger than the largest size class, Carbink rounds the object size up to the nearest 8KB-aligned size, and allocates a dedicated span to hold it.

To allocate spans locally, Carbink uses a *local page heap*. The page heap is an array of free lists, with each list tracking 8KB-aligned free spans of a particular size (e.g., 2MB, 4MB, etc.). If Carbink cannot find a free span big enough to satisfy an allocation request, Carbink allocates a new span, using `mmap()` to request 2MB huge pages from the OS.

Allocating and deallocating via the page heap is mutex-protected because application threads may issue concurrent allocations or deallocations. To reduce contention on the page heap, each thread reserves a private (i.e., thread-local) cache of free spans for each size class. Carbink also maintains a global cache of free lists, with each list having its own spinlock. When a thread wants to allocate a span whose size can be handled by one of Carbink's predefined size classes, the thread first tries to allocate from the thread-local cache, then the global cache, and finally the page heap. For larger allocation requests, threads allocate spans directly from the page heap.

Carbink associates each span with several pieces of metadata, including an integer that describes the span's size class, and a bitvector that indicates which object slots are free. To map a locally-resident object to its associated span metadata, Carbink uses a two-level radix tree called the *local page map*. The lookup procedure is similar to a page table walk: the first 20 bits of an object's virtual address index into the first-level radix tree table, and the next 15 bits index into a second-level table. The same mapping approach allows Carbink to map the virtual address of a locally-resident span to its metadata.

Far memory management: On a compute node, local spans contain a subset of an application's memory state. The rest of that state is stored in far spans that live in far memory regions. Recall from Figure 2b that a Carbink pointer to a non-local object embeds the object's Region ID and Span ID.

To allocate or deallocate a region, a compute node sends a request to the memory manager. A single Carbink region is 1GB or larger, since Carbink targets applications whose total memory requirements are hundreds or thousands of GBs. Upon successfully allocating a region, the compute node updates a *region table* which maps the Region ID of the allocated region to the associated far memory node.

A compute node manages far spans and far regions using additional data structures that are analogous to the ones that manage local spans. A *far page heap* handles the allocation and deallocation of far spans belonging to allocated regions. A *far page map* associates a far Span ID with metadata that (1) names the enclosing region (as a Region ID) and (2) describes the offset of the far span within that region.

Each application thread has a private far cache; Carbink also maintains a global far cache that is visible to all application threads. To swap out a local span of size *s*, a compute node must first use the far page heap (or a far cache if possible) to allocate a free far span of size *s*. Similarly, after a compute node swaps in a far span, the node deallocates the far span, returning the far span to its source (either the far page heap or a far cache).

Span filtering and swapping: The Carbink runtime executes *filtering threads* that iterate through the objects in locally-resident spans and move those objects to different local spans. Carbink's object shuffling has two goals.

- First, Carbink wants to create *hot spans* (containing only hot objects) and *cold spans* (containing only cold ones); when local memory pressure is high, Carbink's *eviction threads* prefer to swap out spansets containing cold spans. Carbink tracks object hotness using GC-style read/write barriers [4, 23]. Thus, by the time that a filtering thread examines an object, the Hotness byte in the object's pointer (see Figure 2) has already been set. Upon examining the Hotness byte, a filtering thread updates the byte using the CLOCK algorithm [12].
- Second, object shuffling allows Carbink to garbage-collect dead objects by moving live objects to new spans and then deallocating the old spans. During eviction, Carbink utilizes efficient one-sided RMA writes to swap spansets out to far memory nodes; this approach allows Carbink to avoid software-level overheads (e.g., associated with thread scheduling) on the far node.

From the application's perspective, object movement and spanset eviction are transparent. This transparency is possible because each object embeds a reverse pointer (§3.2) that allows filtering threads and evicting threads to determine which smart pointers require updating.

Carbink swaps far memory into local memory at the granularity of a span. As with swap-outs, Carbink uses one-sided RMAs for swap-ins. Swapping at the granularity of a span simplifies far memory management, since compute nodes only have to remember how spans map to memory nodes (as opposed to how the much larger number of *objects* map to memory nodes). However, swapping in at span granularity instead of object granularity has a potential disadvantage: if a compute node swaps in a span containing multiple objects, but only uses a small number of those objects, then the compute node will have wasted network bandwidth (to fetch the unneeded objects) and CPU time (to update the remotable pointers for those unneeded objects). We collectively refer to these penalties as *swap-in amplification*.

To reduce the likelihood of swap-in amplification, Carbink's filtering and eviction threads prioritize the scanning and eviction of spansets containing large objects. The associated spans contain fewer objects per span; thus, swapping in these spans will reduce the expected number of unneeded objects. Figure 3 shows that, for our production workloads, large objects occupy the majority of memory. Moreover, most hot objects are small; for example, in our company's geo-distributed database [13], roughly 95% of accesses involve objects smaller than 1.8KB. As a result, an eviction scheme which prioritizes large-object spansets is well-suited for our target applications.

In Carbink, a local span has a three-state lifecycle. A span is first *created* due to a swap-in or local allocation. The span transitions to the *filtering* state upon being examined by filtering threads. Once filtering completes, those spans transition to the *evicting* state when evicting threads begin to swap out spansets. The transition from created to filtering to evicting is fixed, and determines which Carbink runtime threads race with application threads at any given moment (§3.6).

3.4 Fault Tolerance via Erasure Coding

Erasure coding provides data redundancy with lower storage overhead than traditional replication. However, the design space for erasure coding schemes is more complex. Carbink seeks to minimize both average and long-tail access penalties for far objects; per our fault model (§3.1), Carbink also wants to efficiently recover from the failure of memory nodes. Achieving these goals forced us to make careful decisions involving coding granularity, parity recalculation, and cross-node transport protocols.

Coding granularity: To motivate Carbink's decision to erasure-code at the spanset granularity, first consider an approach that erasure-codes individual spans. In this approach, to swap out a span, a compute node breaks the span into data fragments, generates the associated parity fragments, and then writes the entire set of fragments (data+parity) to remote nodes. During the swap-in of a span, a compute node must fetch multiple fragments to reconstruct the target span.

This scheme, which we call EC-Split, is used by Hydra [29]. With EC-Split, handling the failure of memory nodes during swap-out or swap-in is straightforward: the compute node who is orchestrating the swap-out or swap-in will detect the memory node failure, select a replacement memory node, trigger span reconstruction, and then restart the swap-in or

Schemes	EC data fragment size	Network transport	Parity computation	Defragmentation
EC-Split (Hydra [29])	Span chunk	RMA in & out	Local	N/A
EC-2PC	Full span	RMA in, RPC out (+updating parity via 2PC)	Remote	N/A
EC-Batch Local (**Carbink**)	Full span	RMA in & out	Local	Remote compaction
EC-Batch Remote (**Carbink**)	Full span	RMA in & out (+parallel 2PC for compaction)	Local (swap-out)+ Remote (compaction)	Remote compaction

Table 1: The erasure-coding approaches that we study.

swap-out. The disadvantage of EC-Split is that, to reconstruct a single span, a compute node must contact multiple memory nodes to pull in all of the needed fragments. This requirement to contact multiple memory nodes makes the swap-in operation vulnerable to stragglers (and thus high tail latency[2]). This requirement also frequently prevents a compute node from offloading computation to memory nodes; unless a particular object is small, the object will span multiple fragments, meaning that no single memory node will have a complete local copy of the object.

An alternate approach is to erasure-code across a group of equal-sized spans. We call such a group a *spanset*. In this approach, each span in the spanset is treated as a fragment, with parity data computed across all of the spans in the set. To reconstruct a span, a compute node merely has to contact the single memory node which stores the span. Carbink uses this approach to minimize tail latencies.

Parity updating: Erasure-coding at the spanset granularity but swapping in at the span granularity does introduce complications involving parity updates. The reason is that swapping in a span s leaves an invalid, span-sized hole in the backing spanset; the hole must be marked as invalid because, when s is later swapped out, s will be swapped out as part of a new spanset. The hole created by swapping in s causes fragmentation in the backing spanset. Determining how to garbage-collect the hole and update the relevant parity information is non-trivial. Ideally, a scheme for garbage collection and parity updating would not incur overhead on the critical path of swap-ins or swap-outs. An ideal scheme would also allow parity recalculations to occur at either compute nodes or memory nodes, to enable opportunistic exploitation of free CPU resources on both types of nodes.

Cross-node transport protocols: In systems like RAMCloud [39], machines use RPCs to communicate. RPCs involve software-level overheads on both sides of a communication. Carbink avoids these overheads by using one-sided RMA, avoiding unnecessary thread wakeups on the receiver. However, in and of itself, RMA does not automatically solve the consistency issues that arise when offloading parity calculations to remote nodes (§3.4.2).

Throughout the paper, we compare Carbink's erasure-coding approach to various alternatives.

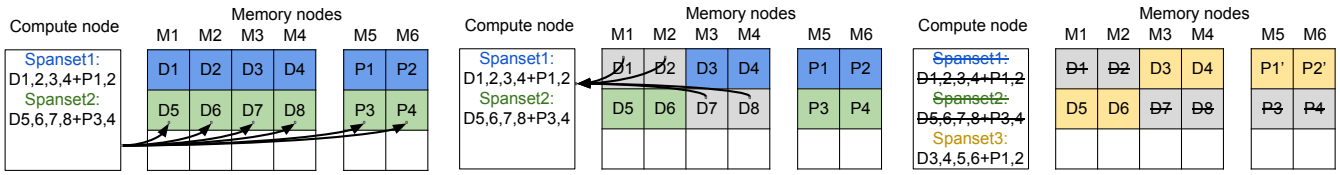
[2]Hydra [29] and EC-Cache [42] try to minimize straggler-induced latencies by contacting $k+\Delta$ memory nodes instead of the minimum k, using the first k responses to reconstruct an object. This approach increases network traffic and compute-node CPU overheads.

- **EC-Split** is Hydra's approach, which erasure-codes at the span granularity, swaps data using RMA, and synchronously recalculates parity at compute nodes when swap-outs occur. Fragmentation within an erasure-coding group never occurs, as a span is swapped in and out as a full unit.
- **EC-2PC** erasure-codes using spansets, and uses RMA to swap in at the span granularity. During a swap-out (which happens at the granularity of a span), EC-2PC writes the updated span to the backing memory node; the memory node then calculates the updates to the parity fragments, and sends the updates to the relevant memory nodes which store the parity fragments. To provide crash consistency for the update to the span and the parity fragments, EC-2PC implements a two-phase commit protocol using RPCs. There is no fragmentation within an erasure-coding group because swap-ins and swap-outs both occur at the span granularity.
- **EC-Batch Local** and **EC-Batch Remote** are the approaches used by Carbink. Both schemes erasure-code at spanset granularity, using RMA for swap-in as well as swap-out. Swap-ins occur at the granularity of a span, but swap-outs occur at the granularity of spansets (§3.4.1); thus, both EC-Batch approaches deallocate a span's backing area in far memory upon swapping that span into a compute node's local RAM. The result is that swap-ins create dead space on a remote memory node. Both EC-Batch schemes reclaim dead space and recalculate parity data using asynchronous garbage collection. EC-Batch Local always recalculates parity on compute nodes, whereas EC-Batch Remote can recalculate parity on compute nodes or memory nodes. When EC-Batch Remote offloads parity computations to remote nodes, it employs a parallel commit scheme that avoids the latencies of traditional two-phase commit (§3.4.2).

Table 1 summarizes the various schemes. We now discuss EC-Batch Local and Remote in more detail.

3.4.1 EC-Batch: Swapping

Swapping out: In both varieties of EC-Batch, a spanset contains multiple spans of the same size. At swap-out time, a compute node writes a *batch* (i.e., a spanset and its parity fragments) to a memory node. Figure 4a shows an example. In that example, the compute node has two spansets: spanset1 (consisting of data spans $< D1, D2, D3, D4 >$ and parity fragments $< P1, P2 >$), and spanset2 (containing data spans $< D5, D6, D7, D8 >$ and parity fragments $< P3, P4 >$). Carbink uses Reed-Solomon codes [43] to create parity data, and prioritizes the eviction of spansets that contain cold spans



(a) Swapping out spans and parity in a batch. **(b)** Swapping in individual spans. **(c)** Compacting spansets to reclaim space.

Figure 4: EC-Batch swapping-out, swapping-in, and far compaction.

(§3.3). Neither variant of EC-Batch updates spansets in place, so eviction may require a compute node to request additional far memory regions from the memory manager.

Swapping in: When an application tries to access an object that is currently far, the Carbink runtime inspects the application pointer and extracts the Span ID (see Figure 2b). The runtime consults the far page map (§3.3) to discover which remote node holds the span. Finally, the runtime initiates the appropriate RMA operation to swap in the span.

However, swapping in at the span granularity creates *remote fragmentation*. In Figure 4b, the compute node in the running example has pulled four spans ($D1$, $D2$, $D7$, and $D8$) into local memory. Any particular span lives exclusively in local memory or far memory; thus, the swap-ins of the four spans creates dead space on the associated remote memory nodes. If Carbink wants to fill (say) $D1$'s dead space with a new span $D9$, Carbink must update parity fragments $P1$ and $P2$. For a Reed-Solomon code, those parity fragments will depend on both $D1$ and $D9$.

There are two strawman approaches to update $P1$ and $P2$:

- The compute node can read $D1$ into local memory, generate the parity information, and then issue writes to $P1$ and $P2$.
- Alternatively, the compute node can send $D9$ to memory node $M1$, and request that $M1$ compute the new parity data and update $P1$ and $P2$.

The second approach requires a protocol like 2PC to guarantee the consistency of data fragments and parity fragments; without such a protocol, if $M1$ fails after updating $P1$, but before updating $P2$, the parity information will be out-of-sync with the data fragments.

The first approach, in which the compute node orchestrates the parity update, avoids the inconsistency challenges of the second approach. If a memory node dies in the midst of a parity update, the compute node will detect the failure, pick a new memory node to back the parity fragment, and retry the parity update. If the compute node dies in the midst of the parity update, then the memory manager will simply deallocate all regions belonging to the compute node (§3.1).

Unfortunately, both approaches require a lot of network bandwidth to fill holes in far memory. To reclaim one vacant span, the first approach requires three span-sized transfers—the compute node must read $D1$ and then write $P1$ and $P2$. The second approach requires two span-sized transfers to update $P1$ and $P2$. To reduce these network overheads, Carbink performs *remote compaction*, as described in the next section.

3.4.2 EC-Batch: Remote Compaction

Carbink employs *remote compaction* to defragment far memory using fewer network resources than the two strawmen above. On a compute node, Carbink executes several *compaction threads*. These threads look for "matched" spanset pairs; in each pair, the span positions containing dead space in one set are occupied in the other set, and vice versa. For example, the two spansets in Figure 4b are a matched pair. Once the compaction threads find a matched pair, they create a new spanset whose data consists of the live spans in the matched pair (e.g., $< D3, D4, D5, D6 >$ in Figure 4b). The compaction threads recompute and update the parity fragments $P1'$ and $P2'$ using techniques that we discuss in the next paragraph. Finally, the compaction threads deallocate the dead spaces in the matched pair (e.g., $< D1, D2, D7, D9, P3, P4 >$ in Figure 4b), resulting in a situation like the one shown in Figure 4c. Carbink's compaction can occur in the background, unlike the synchronous parity updates of EC-2PC which place consensus activity on the critical path of swap-outs.

So, how should compaction threads update parity information? Carbink uses Reed-Solomon codes over the Galois field $GF(2^8)$. The new parity data to compute in Figure 4c is therefore represented by the following equations on $GF(2^8)$:

$$P1' - P1 = A_{1,1}(D5 - D1) + A_{2,1}(D6 - D2)$$
$$P2' - P2 = A_{1,2}(D5 - D1) + A_{2,2}(D6 - D2)$$

where $A_{i,j}$ $(i \in \{0,1,2,3\}, j \in \{0,1\})$ are fixed coefficient vectors in the Reed-Solomon code. Carbink provides two approaches for updating the parity information.

- In EC-Batch Local, the compute node that triggered the swap-out orchestrates the updating of parity data. In the running example, the compute node asks $M1$ to calculate the span delta $D5 - D1$, and asks $M2$ to calculate the span delta $D6 - D2$. After retrieving those updates, the compute node determines the parity deltas (i.e., $P1' - P1$ and $P2' - P2$) and pushes those deltas to the parity nodes $M5$ and $M6$.
- In EC-Batch Remote, the compute node offloads parity recalculation and updating to memory nodes. In the running example, the compute node asks $M1$ to calculate the span delta $D5 - D1$, and $M2$ to calculate the span delta $D6 - D2$. The compute node also asks $M1$ and $M2$ to calculate partial parity updates (e.g., $A_{1,1}(D5 - D1)$ and $A_{1,2}(D5 - D1)$ on $M1$). $M1$ and $M2$ are then responsible for sending the partial parity updates to the parity nodes. For example, $M1$ sends $A_{1,1}(D5 - D1)$ to $M5$, and $A_{1,2}(D5 - D1)$ to $M6$.

In EC-Batch Local, recovery from memory node failure is orchestrated by the compute node in a straightforward way, as in EC-Split (§3.4). In EC-Batch Remote, a compute node performs remote compaction by offloading parity updates to memory nodes. The compute node ensures fault tolerance for an individual compaction via 2PC. However, the compute node aggressively issues compaction requests in parallel. Two compactions (i.e., two instance of the 2PC protocol) are safe to concurrently execute if the compactions involve different spansets; the prepare and commit phases of the two compactions can partially or fully overlap.

On a compute node, Carbink's runtime can monitor the CPU load and network utilization of remote memory nodes. The runtime can default to remote compaction via EC-Batch Local, but opportunistically switch to EC-Batch Remote if spare resources emerge on memory nodes. During a switch to a different compaction mode, Carbink allows all in-flight compactions to complete before issuing new compactions that use the new compaction mode.

The strawmen defragmentation schemes in Section 3.4.1 require two or three span-sized network transfers to recover one dead span. In the context of Figure 4, EC-Batch Local recovers four dead spans using four span-sized network transfers. EC-Batch Remote requires four span-sized network transfers (plus some small messages generated by the consistency protocol) to recover four dead spans.

3.5 Failure Recovery

Carbink handles two kinds of memory node failures: planned and unplanned. Planned failures are scheduled by the cluster manager [51, 52] to allow for software updates, disk reformatting, and so on. Unplanned failures happen unexpectedly, and are caused by phenomena like kernel panics, defective hardware, and power disruptions.

Planned failures: When the cluster manager decides to schedule a planned failure, the manager sends a warning notification to the affected memory nodes. When a memory node receives such a warning, the memory node informs the memory manager. In turn, the memory manager notifies any compute nodes that have allocated regions belonging to the soon-to-be-offline memory node. Those compute nodes stop swapping-out to the memory node, but may continue to swap-in from the node as long as the node is still alive. Meanwhile, the memory manager orchestrates the migration of regions from the soon-to-be-offline memory node to other memory nodes. When a particular region's migration has completed, the memory manager informs the relevant compute node, who then updates the local mapping from Region ID to backing memory node. At some point during this process, the memory manager may also request non-failing memory nodes to contribute additional regions to the global pool of far memory.

Unplanned Failures: On a compute node, the Carbink runtime is responsible for detecting the unplanned failure of a memory node. The runtime does so via connection timeouts or more sophisticated leasing protocols [15, 16]. Upon detecting an unplanned failure, the runtime spawns background threads to reconstruct the affected spans using erasure coding. The runtime is also responsible for allowing application threads to read spans whose recovery is in-flight.

Span reconstruction: To reconstruct the spans belonging to a failed memory node M_{fail}, a compute node first requests a new region from the memory manager. Suppose that the new region is provided by memory node M_{new}. The compute node iterates through each lost spanset associated with M_{fail}; for each spanset, the compute node tells M_{new} which external spans and parity fragments to read in order to erasure-code-restore M_{fail}'s data. As the relevant spans are restored, a compute node can still swap in and remotely compact those spans. However, the swap-in and remote compaction activity will have to synchronize with recovery activity (§3.6).

In EC-Batch Local, when a compute node detects a memory node failure, the compute node cancels all in-flight compactions involving that node. A compute node using EC-Batch Remote does the same; however, for each canceled compaction, the compute node must also instruct the surviving memory nodes in the 2PC group to cancel the transaction.

The data and parity for a swapped-out spanset reside on multiple memory nodes. As a compute node recovers from the failure of one of the nodes in that group, another node in the group may fail. As long as the number of failed nodes does not exceed the number of parity nodes, Carbink can recover the spanset. The reason is that all of the information needed to recover is stored on a compute node, e.g., in the far page heap (§3.3). Due to space limitations, we omit a detailed explanation of how Carbink deals with concurrent failures.

Degraded reads: During the reconstruction of an affected span, application threads may try to swap in the span. The runtime handles such a fetch using a *degraded read* protocol. For example, consider Figure 4a. Suppose that $M1$ fails unexpectedly, and while the Carbink runtime is recovering $M1$'s spans ($D1$ and $D5$), an application thread tries to read an object residing in $D1$. The runtime will swap in data spans $D2$, $D3$, and $D4$, as well as parity fragment $P1$, and then reconstruct $D1$ via erasure coding. Degraded reads ensure that the failure of a memory node merely slows down an application instead of blocking it. In Section 5.3, we show that application performance only drops for 0.6 seconds, and only suffers a throughput degradation of 36% during that time.

Network bandwidth consumption: During failure recovery, Carbink consumes the same amount of network bandwidth as Hydra. For example, suppose that both Hydra and Carbink use RS4.2 encoding and have 4 spans, with a span stored on each of 4 memory nodes. In Hydra, a single node failure will lose four 1/4th spans. Reconstructing each 1/4th span will require the reading of four 1/4th span/parity regions from the surviving nodes, resulting in an aggregate network bandwidth requirement of 1 full span. So, reconstructing four 1/4th spans

will require an aggregate network bandwidth of 4 full spans. In Carbink, the failure of a single memory node results in the loss of 1 full span. To recover that span, Carbink (like Hydra) must read 4 span/parity regions.

3.6 Thread Synchronization

On a compute node, the main kinds of Carbink threads are applications threads (which read objects, write objects, and swap in spans), filtering threads (which move objects within local spans), and eviction threads (which reclaim space by swapping local spansets to far memory). At any given time, a span may be in one of two concurrency regimes (§3.3): the span is either accessible to application threads and filtering threads, or to application threads and eviction threads. In both regimes, Carbink has to synchronize how the relevant threads update Carbink's smart pointers (§3.2).

At a high level, Carbink uses an RCU locking scheme that is somewhat reminiscent of AIFM's approach [44]. Due to space restrictions, we merely sketch the design. Carbink optimizes for the common case in which a span is only being accessed by an application thread. In this common case, an application thread grabs an RCU read lock on the pointer via the pointer's `Deref()` method, as shown in Listing 1. The thread sees that either (1) the **P**resent bit is not set, in which case the Carbink runtime issues an RMA read to swap in the appropriate span; (2) alternatively, the thread sees that the **P**resent bit is set, but the **M** and **E** bits are unset. In the second case, `Deref()` can just return a normal pointer back to the application. The application can be confident that concurrent filtering or evicting threads will not move or evict the object, because those threads cannot touch the object until application-level threads have released their RCU read locks via the `DerefScope` destructor (Listing 1).

The more complicated scenarios arise when the **P**resent bit is set and either the **M** or **E** bit are set as well. In this case, the (say) **M** bit has been set because the filtering thread set the bit and then called `SyncRCU()` (i.e., the RCU write waiting lock). The concurrent application thread and filtering thread essentially race to acquire the pointer's spinlock; if the application thread (i.e., `Deref()`) wins, it makes a copy of the object, clears **M**, releases the spinlock, and returns the address of the object copy to the application. Otherwise, if the filtering thread wins, it moves the object, clears **M**, and releases the spinlock. The losing thread has to retry the desired action. An analogous situation occurs if the **E** bit is set.

Carbink's eviction and remote compaction threads directly poll the network stack to learn about RMA completions and RPC completions. An application thread which has issued an RMA swap-in operation will yield, but a dedicated RMA poller thread detects when application RMAs have completed and awakens the relevant application threads. Polling avoids the overheads of context switching to new threads and notifying old threads that network events have occurred.

During recovery (§3.5), Carbink spawns additional threads to orchestrate the reconstruction of spans. Those threads acquire per-spanset mutexes which are also acquired by threads performing swap-ins, swap-outs, and remote compactions.

4 Implementation

Our Carbink prototype contains 14.3K lines of C++. It runs atop unmodified OSes, using standard POSIX abstractions for kernel-visible threads and synchronization. The runtime leverages the PonyExpress user-space network stack [35]. On a compute node, all threads in a particular application (both application-defined threads and Carbink-defined threads) execute in the same process. On a memory node, a Carbink daemon exposes far memory via RMAs or RPCs. We use Intel ISA-L v2.30.0 [25] for Reed-Solomon erasure coding.

Our current prototype has a simplified memory manager that is unreplicated, does not handle planned failures, and statically assigns memory nodes to compute nodes. Implementing the full version of the memory manager will be conceptually straightforward, since we can use off-the-shelf libraries for replicated state machines [1, 45] and cluster management [51, 52]. We also note that the experiments in §5 are insensitive to the performance of the memory manager, regardless of whether the manager is replicated or not. The reason is that memory allocations and deallocations (which must be routed through the memory manager) are rare and are not on the critical path of steady-state compute node operations like swap-in and swap-out.

To better understand the performance overheads of Carbink's erasure-coding approach, we built an AIFM-like [44] far memory system. That system uses remotable pointers like Carbink, but swaps in and out at the granularity of objects, and provides no fault tolerance. Like Carbink, it leverages the PonyExpress [35] user-space network stack. Our AIFM clone is 5.8K lines of C++.

5 Evaluation

In this section, we answer the following questions:

1. What is the latency, throughput, and remote memory usage of EC-Batch compared with the other fault tolerance schemes (§5.1 and §5.2)?
2. How does an unplanned memory node failure impact the performance of Carbink applications (§5.3)?
3. How does the performance of Carbink's span-based memory organization compare to the performance of an AIFM-like object-level approach (§5.4)?

Testbed setup: We deployed eight machines in the same rack, including one compute node and seven memory nodes; one of the memory nodes was used for failover. Each machine was equipped with dual-socket 2.2 GHz Intel Broadwell processors and a 50 Gbps NIC.

Fault tolerance schemes: Using the Carbink runtime, we compared our proposed EC-Batch schemes to four ap-

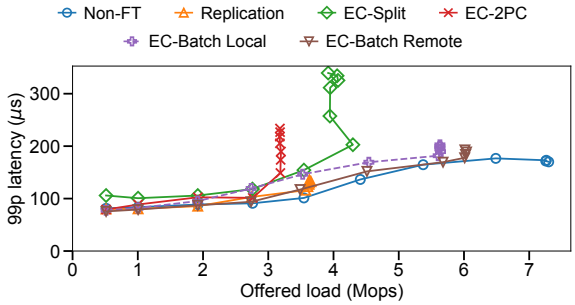


Figure 5: Microbenchmark load-latency curves.

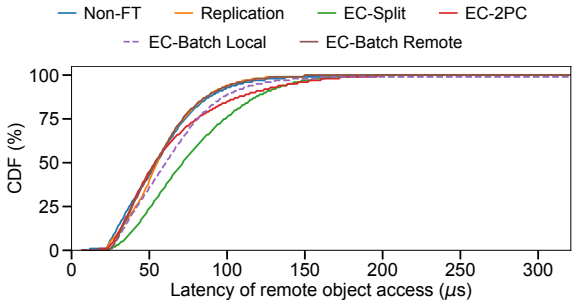


Figure 6: Latency distribution of remote object accesses in the microbenchmark under an offered load of 2 Mops.

proaches: Non-FT (a non-fault-tolerant scheme that used RMA to swap spans), Replication (which replicated spans on multiple nodes), EC-Split (the approach used by Hydra [29]), and EC-2PC (Table 1). We configured all fault tolerance schemes to tolerate up to two memory node failures. So, the Replication scheme replicated each swapped-out span on three memory nodes, whereas the EC schemes used six memory nodes—four held data, and two held RS4.2 parity bits [43]. EC-Batch spawned two compaction threads by default.

As mentioned in Section 4, we also built an AIFM-like far memory system. This system did not provide fault tolerance, but it provided a useful comparison with our Non-FT Carbink version.

Carbink borrows the span sizes that are used by TCMalloc (§3.3). These parameters have been empirically observed to reduce internal fragmentation. In our evaluation, EC-Batch (both Local and Remote) grouped four equal-size spans into a spanset, swapping out at the granularity of a spanset. Increasing spanset sizes would allow Carbink to issue larger batched RMAs, improving network efficiency. However, spansets whose evictions are in progress must be locked in local memory while RMAs complete; thus, larger spanset sizes would delay the reclamation of larger portions of local memory.

5.1 Microbenchmarks

To get a preliminary idea of Carbink's performance, we created a synthetic benchmark that wrote 15 million 1 KB objects (totalling 15 GB) to a remotable array. The compute node's local memory had space to store 7.5 GB of objects (i.e., half of the total set). By default, the compute node spawned 128

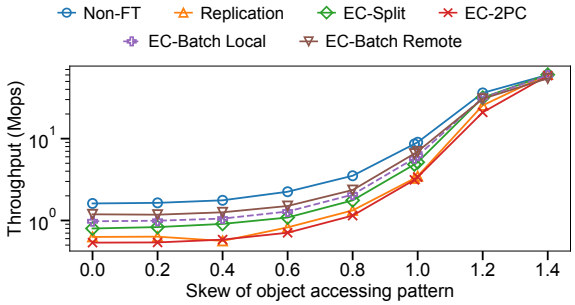


Figure 7: Impact of skew on throughput.

threads on 32 logical cores to access objects; the access pattern had a Zipfian-distributed [41] skew of 0.99. Such skews are common in real workloads for key/value stores [7].

Object access throughput and tail latency: Figure 5 shows the 99th-percentile latency with various object access loads. All of the fault-tolerant schemes eventually hit a "hockey stick" in tail latency growth when the schemes could no longer catch up with the offered load. EC-Batch Remote had the highest sustained throughput (6.0 Mops), which was 40% higher than the throughput of the state-of-the-art EC-Split (4.3 Mops). EC-Batch Local achieved 5.6 Mops, which was 30% higher than EC-Split. EC-Split had worse performance because it had to issue four RMA requests to swap in one span; thus, EC-Split quickly became bottlenecked by network IO. In contrast, EC-Batch only issued one RMA request per swap-in.

EC-Batch Remote had 18%-29% lower tail latency than EC-Split under the same load (before reaching the "hockey-stick"). The reason was that EC-Split's larger number of RMAs per swap-in left EC-Split more vulnerable to stragglers [29]. Also recall that EC-Batch can support computation offloading [3, 27, 44, 57], which is hard with EC-Split (§3.4).

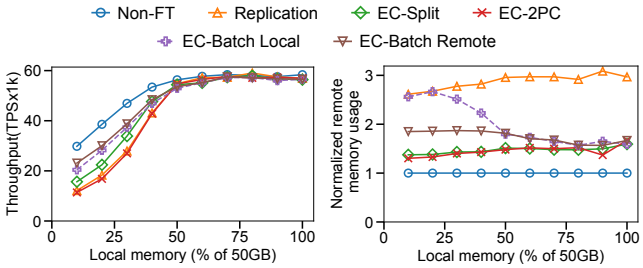
EC-2PC had the worst throughput because it relied on costly RPCs and 2PC protocols to swap out spans. Thus, EC-2PC could not reclaim local memory as fast as other schemes. The Replication scheme was bottlenecked by network bandwidth, since every swap-out incurred a 3× network write penalty; in contrast, EC-based schemes used RS4.2 erasure coding to reduce the write penalty to 1.5×.

Latency distribution of remote object accesses: Figure 6 shows the latency of accessing remote objects under 2 Mops of offered load. With this low offered load, Replication and EC-Batch Remote achieved similar access latencies as Non-FT because none of the schemes were bottlenecked by network bandwidth. EC-Batch Local had slightly higher remote access latencies. However, EC-Split had significantly higher access latencies (e.g., at the median and tail) than EC-Batch Local and Remote; the reason was that EC-Split issued four times as many network IOs and thus was more sensitive to stragglers. EC-2PC's tail latency was slightly higher than that of EC-Batch Local and Remote due to the overhead of costly RPCs and 2PC traffic.

Impact of skewness: Figure 7 shows how the skewness of object accesses impacted throughput. EC-Batch Remote and

	# Compaction threads	Norm. remote mem usage	Avg. # remote logical cores	Avg. BW (Gbps)
EC-Batch Local	1	2.54	0.23	1.27
	2	2.35	0.53	1.64
	3	2.28	0.56	1.76
EC-Batch Remote	1	1.89	1.97	2.98
	2	1.83	2.10	3.15
	3	1.74	2.27	3.40
W/o compaction	0	3.03	–	–

Table 2: Remote resource usage in the microbenchmark. The remote memory usage is normalized with respect to the usage of Non-FT. The number of remote logical cores and the network bandwidth are averaged across all six memory nodes.



(a) Transaction throughput. **(b)** Remote memory usage.

Figure 8: Transactional KV-store evaluation.

Local performed best due to their more efficient swapping approaches. However, the throughput of all schemes increased with higher skewness. The reason is that high skewness led to a smaller working set and thus a higher likelihood that hot objects were locally resident. In these scenarios, schemes with faster swapping were not rewarded as much.

Remote resource usage with compaction: Table 2 shows the impact of compaction on the average memory, CPU, and bandwidth usage per memory node. Without compaction, EC-Batch used 3.03× remote memory (normalized with respect to Non-FT memory consumption). With two local compaction threads, EC-Batch Remote's memory overhead reduced to 1.83×. The memory reduction was at the expense of 2.1 cores and 3.15 Gbps bandwidth on each memory node. With more compaction threads, Carbink could further reduce memory usage at the cost of higher CPU and bandwidth utilization. That being said, we note that the synthetic microbenchmark application represented an extreme case of remote CPU and network usage, since the workload accessed objects without actually computing on them.

EC-Batch Remote vs. Local: EC-Batch Remote had higher throughput and lower tail latency than EC-Batch Local (Figure 5). This was because EC-Batch Local's compaction required (1) local CPUs for parity computation and (2) network bandwidth for transferring span deltas and parity updates, leaving fewer local resources for application threads and RMA reads. Because of EC-Batch Remote's faster compaction, EC-Batch Remote also used 28%-34% less remote memory than EC-Batch Local (Table 2). However, EC-Batch Remote consumed more remote CPUs (2.10 vs. 0.53 cores) and more network bandwidth (3.15 vs. 1.64 Gbps) than Local. In practice, the Carbink runtime could transparently switch between EC-Batch Remote and Local based on an application developer's policy about resource/performance trade-offs.

5.2 Macrobenchmarks

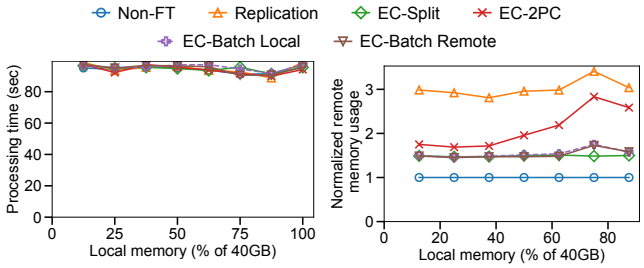
We evaluated Carbink using two memory-intensive applications that would benefit from remote memory: an in-memory transactional key-value store, and a graph processing algorithm. The two applications exhibited different patterns of object accesses, and had different working set behaviors.

Transactional KV-store: This application implemented a transactional in-memory B-tree, exposing it via a key/value interface similar to that of MongoDB [37]. Each remotable object was a 4 KB value stored in a B-tree leaf. The application spawned 128 threads, and each thread processed 20 K transactions. The compute node provisioned 32 logical cores, with the application overlapping execution of the threads for higher throughput [26, 38, 44, 56]. Each transaction contained three reads and three writes, similar to the TPC-A benchmark [53]. Each update created a new version of a particular key's value; asynchronously, the application trimmed old versions. The maximum working set size during the experiment was roughly 50 GB.

Throughput: Figure 8a shows the KV-store throughput when varying the size of local memory (normalized as a fraction of the maximum working set size). In scenarios with less than 50% local memory, EC-Batch Remote achieved higher transactions per second (TPS) than all other fault tolerance schemes. For example, TPS for EC-Batch Remote was 1.5%-48% higher than that of EC-Split; this was because EC-Batch only needed one RMA request to swap in a span. EC-Batch Remote was at most 29% slower than Non-FT, mainly due to the additional parity update required for fault tolerance. EC-Batch Local was at most 13% slower than EC-Batch Remote. EC-2PC performed the worst among EC schemes.

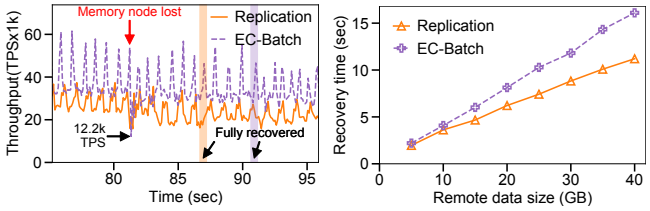
All schemes achieved similar throughput when the local memory size was above 50%. The reason was that the *average* working set size of the workload was only half the size of the *maximum* memory usage. The maximum memory usage only occurred when the B-Tree had fallen very behind in culling old versions of objects.

Remote memory usage: Figure 8b plots remote memory usage as a function of local memory sizes; remote memory usage is normalized with respect to that of Non-FT. Compared to EC-Split, EC-Batch Remote and Local used up to 35% and 93% more remote memory, respectively. EC-Batch schemes defragmented remote memory using compaction, but when local memory space was less than 50%, remote compaction could not immediately defragment the spanset holes created by frequent span swap-ins. As local memory grew larger, span fetching became less frequent, making it



(a) Processing time. **(b)** Remote memory usage.

Figure 9: Graph processing evaluation.



(a) KV-store TPS over time. **(b)** Microbenchmark recovery.

Figure 10: Failure recovery evaluation.

easier for remote compaction to reclaim space. In this less hectic environment, EC-Batch's remote memory usage was similar to that of the other erasure-coding schemes.[3]

Graph processing: We implemented a connected-components algorithm [50] that found all sets of linked vertices in a graph. This kind of algorithm is critical to various Google services. We evaluated the algorithm using the Friendster graph [30] which contained 65 million vertexes and 1.8 billion edges. In the graph analysis code, each vertex's adjacency list was referenced via remotable pointers. The total size of the objects stored in Carbink was roughly 40 GB. The application used 80 application threads that ran atop 80 logical cores. In our experimental results, the reported processing times exclude graph loading, since graph loading is dominated by disk latencies.

Figure 9a shows that all schemes had similar processing times as Non-FT, regardless of the local memory size. The reason was that the graph application had a high compute-to-network ratio—the application fetched all neighbors associated with each vertex and then spent non-trivial time enumerating each neighbor and computing on them. As a result of this good spatial locality and high "think time," the graph application did not incur frequent data swapping, and thus avoided fault tolerance overhead that the KV-store could not.

Figure 9b shows that EC-Batch Local and Remote had similar remote memory usage as EC-Split: 15%-39% lower than EC-2PC and roughly 50% lower than Replication. All EC-based schemes had lower remote memory overheads than Replication because the erasure coding only incurred a 1.5× space overhead for the extra parity data.

EC-2PC used more memory than EC-Batch because the graph workload randomly fetched diverse-sized spans. The random fetch sizes reflected the fact that different vertices had different sizes for their adjacency lists. This lack of span size locality hindered dead space reclamation, since EC-2PC had to wait longer for all of the spans in an erasure-coding group to be swapped in. EC-Batch avoided this problem by bundling equal-sized spans into the same spanset and using remote compaction.

[3]The remote memory usage of triple-replication was slightly less than 3× the usage of Non-FT because Non-FT could swap out memory faster during periods of high local memory pressure.

5.3 Failure Recovery

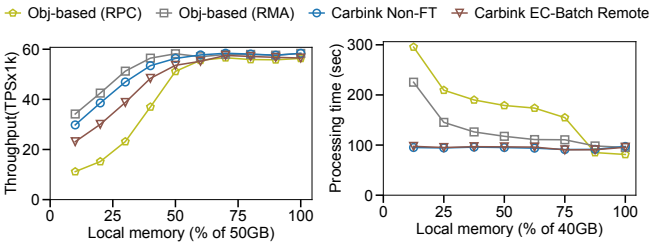
We measured the recovery time for an unplanned memory node failure in the KV-store, the graph processor, and the microbenchmark application. For the graph application, all schemes achieved similar processing time during unplanned failures; thus, in the text below, we focus on the KV-store and the microbenchmark.

Transactional KV-store: Figure 10a shows the KV-store throughput of Replication and EC-Batch Local, with a data point collected every 100 ms before and after an unplanned memory node failure. Upon detecting the failure, EC-Batch Local immediately reconstructed the lost data on a pre-configured failover memory node. We gave the KV-store 15 GB of local memory, equivalent to 30% of the 50 GB maximum working set size.

The throughput of both schemes fluctuated sinusoidally because the KV-store frequently tried to swap in remote objects, but the swap-ins sometimes had to synchronously block until eviction threads could reclaim enough local memory. After a memory node failed, EC-Batch needed 0.6 seconds to restore normal throughput, while replication needed 0.3 seconds. This is because, during failure recovery, an EC-Batch read that targeted an affected span used the degraded read protocol which uses more bandwidth than a normal read (§3.5); in contrast, a Replication read that targeted an affected span consumed the same amount of bandwidth as a read during non-failure-recovery. During recovery, the throughput of Replication and EC-Batch dropped an average of 35% and 36% respectively.

EC-Batch required 9.7 seconds to fully regenerate the lost data on the failover node, taking 1.7× longer than Replication. This difference arose because, in EC-Batch, the new memory node read 4× span/parity information involving the lost data and computed erasure codes to reconstruct the lost data. In contrast, Replication lost more data per memory node, but only read one copy of the lost data. Note that with EC-Batch, degraded reads mostly happened during the first second of failure recovery; the skewed workload meant that a small number of objects were the targets of most reads, and once a hot object was pulled into local memory (perhaps by a degraded read), the object would not generate additional degraded reads.

Microbenchmark: Figure 10b shows recovery times as a function of the remote data size. The recovery time of EC-Batch increased almost linearly with the remote data size,



(a) KV-store application. **(b)** Graph application.

Figure 11: Application performance: AIFM-like object-based systems and Carbink.

with 0.6 GB/s recovery speed. This speed was 12%-44% slower than Replication due to the larger amount of recovery information that EC-Batch had to transfer around the network, and the computational overhead of generating erasure codes.

Prior work [10, 29, 58] also found that, during recovery, erasure-coding schemes had longer recovery times and worse performance degradation than replication schemes. However, this drawback only happens for unplanned failures which, in our production environment, are rare compared to planned failures; in an erasure-coding scheme, handling a *planned* failure just requires simple copying of the information on a departing memory node, and does not incur additional work to find parity information or recompute erasure coding. Thus, in our deployment setting where unplanned failures are rare, erasure-coding schemes (which have lower memory utilization than replication schemes) are very attractive.

5.4 Comparison with AIFM-like Systems

We compared span-based swapping in Carbink with the object-based approach used in AIFM [44]. We implemented two AIFM-like systems using our threading and network stack (§4). The first system used RPCs to swap individual objects, with the remote memory nodes tracking the object-to-remote-location mapping (as done in AIFM). Our second object-granularity swapping system used more-efficient RMAs to swap objects, and had compute nodes track the mapping between objects and their remote locations; recall that RMA is one-sided, so compute nodes could not rely on memory nodes to synchronously update mappings during swaps. Like the original AIFM, neither system provided fault tolerance.

Transactional KV-store: Figure 11a shows that, if local memory was too small to hold the average working set, Non-FT Carbink had 45%-167% higher throughput than the AIFM-like system with RPC. The reason is that, when local memory pressure was high, more swapping occurred, and the better efficiency of RMAs over RPCs became important. However, Non-FT Carbink achieved 5.6%-15% lower throughput than the object-based system with RMA. This was due to swap-in amplification. For example, Non-FT Carbink might swap in an 8KB span but only use one 4KB object in the span; this never happens in a system that swaps at an object granularity.

Graph processing: Figure 11b shows the graph application's processing time. When the local memory size was below 87.5%, Carbink performed 18%-58% faster than the object-based system with RMA. This is because, in the graph workload, 4% of large objects occupied 50% of the overall data set. Carbink prioritized swapping out large cold objects (§3.3), keeping most small objects in local memory and reducing the miss rate for those objects. In contrast, the object-based systems did not consider object sizes when swapping, leading to an increased miss rate for small objects. Note that, with larger local memories, all schemes had similar performance; indeed, when all objects fit into local memory, the object-based system with RPC slightly outperformed the rest because it did not require a dedicated core to poll for RMA completions.

6 Discussion

EC-Batch for paging-based systems: Carbink uses EC-Batch to transparently expose far memory via remotable pointers. However, EC-Batch can also be used to expose far memory via OS paging mechanisms [5, 22, 46]. In a traditional paging-based approach for far memory, a compute node swaps in and out at the granularity of a page. However, a compute node can use EC-Batch to treat each page as a span, such that pages are swapped out at the "pageset" granularity, and pages are swapped in at the page granularity.

Custom one-sided operations: EC-Batch requires memory nodes to calculate span deltas and parity updates (§3.4.2). In our Carbink prototype, memory nodes use separate threads to execute these calculations. However, memory nodes could instead implement them as custom one-sided operations in the network stack, such that the network stack itself performs the calculations, avoiding the need to context-switch to external threads. This approach has been used in prior work [6, 9, 35, 47, 48] to avoid thread scheduling overheads.

Designing the memory manager: We used a centralized manager because such a manager (1) simplified our overall design, and (2) made it easier to drive memory utilization high (because a centralized manager will have a global, accurate view of memory allocation metadata). A similarly-centralized memory manager is used by the distributed transaction system FaRM [16]. If the centralized manager became unavailable, Carbink could fall back to a decentralized memory allocation scheme like the one used by Hydra [29] or INFINISWAP [22].

The state maintained by the memory manager is not large. With 1 GB regions, we expect up to 500 regions in a typical memory node (similar to FaRM [16]). With thousands of memory nodes, the memory manager just needs to store a few MBs of state for region assignments.

Fault tolerance for compute nodes: In Carbink, a compute node does not share memory with other compute nodes. Thus, a Carbink application can checkpoint its own state without fear of racing with other compute nodes that modify the state being checkpointed. Checkpoint data could be placed in a

	Fast s/o	Low mem	Fast s/i	Interface	Coding granularity
On-disk rpl.	✗	✓	✓	Various	–
In-memory rpl.	✓	✗	✓	Various	–
Hydra [29]	✓	✓	✗	Paging	Split 4KB pages
Cocytus [10]	✓	✓	✗	KV-store	Across 4KB pages
BCStore [31]	✓	✓	✗	KV-store	Across objs
Hybrid [32]	✗	✗	✓	KV-store	Split 4KB pages
Carbink	✓	✓	✓	Remotable pointers	Across spans

Table 3: Comparison of existing fault-tolerant approaches for far memory. "Fast s/o" indicates whether a system can swap out at network/memory speeds. "Low mem" means that a system has relatively low memory pressure. "Fast s/i" refers to whether a system can swap in at network/memory speeds.

non-Carbink store, obviating the need to track how checkpointed spans move across Carbink memory nodes during compaction and invalidation. Alternatively, Carbink itself could store checkpoints, e.g., in the fault-tolerant address space of a well-known Carbink application whose sole purpose is to store checkpoints.

7 Related Work

Fault tolerance for far memory: Many far memory systems do not provide fault tolerance [2, 44, 55]. Of the systems that do, most replicate swapped-out data to local disks or remote ones [5, 22, 46]. Unfortunately, this approach forces application performance to bottleneck on disk bandwidth or disk IOPs during bursty workloads or failure recovery [29]. This behavior is unattractive, since a primary goal of a far memory system is to have applications run at *memory* speeds as much as possible.

Like Carbink, Hydra [29] is a far memory system that provides fault tolerance by writing erasure-coded local memory data to far RAM. Hydra uses the EC-Split coding approach that we describe in Section 3.4. As we demonstrate in Section 5, Carbink's erasure-coding scheme provides better application performance in exchange for somewhat higher memory consumption. Carbink's coding scheme also enables the offloading of computations to far memory nodes. Such offloading can significantly improve the performance of various applications [3, 27, 44, 57].

Fault tolerance for in-memory transactions and KV-stores: In-memory transaction systems typically provide fault tolerance by replicating data across the memory of multiple nodes [15, 16, 26]. These approaches suffer from the classic disadvantages of replication: double or triple storage overhead, and the associated increase in network traffic.

Recent in-memory KV-stores use erasure coding to provide fault tolerance. For example, Cocytus [10] and BCStore [31] only rely on in-memory replication to store small instances of metadata; object data is erasure-coded using a default page size of 4KB. Cocytus erasure-codes using a scheme that resembles EC-2PC (§3.4). To reduce the network utilization of a Cocytus-style approach, a BCStore compute node buffers outgoing writes; this approach allows the node to batch the computation of parity fragments (and thus issue fewer updates to remote data and parity regions). Batching reduces network overhead at the cost of increasing write latency.

Both Cocytus and BCStore rely on two-sided RPCs to manipulate far memory. RPCs incur software-level overheads involving thread scheduling and context switching on remote nodes. To avoid these costs, Carbink eschews RPCs for one-side RMA operations. Carbink also issues fewer parity updates than Cocytus; whereas Cocytus uses expensive 2PC to update parity information during every write, Carbink defers parity updates until compaction occurs on remote nodes (§3.4.2). Carbink's compaction approach is also more efficient than that of BCStore. BCStore's compaction algorithm performs actual copying of data objects on memory nodes, whereas Carbink compaction just manipulates span pointers inside of spanset metadata.

A far memory system could use both replication and erasure coding [32]. For example, during a Hydra-style swap-out, a span would be erasure-coded and the fragments written to memory nodes; however, a full replica of the span would also be written out. Relative to Carbink, this hybrid approach would have lower reconstruction costs (assuming that the full replica did not live on the failed node). However, Carbink would have lower memory overheads because no full replica of a span would be stored. Carbink would also have faster swap-outs, because swap-outs in the hybrid scheme would require an EC-2PC-like mechanism to ensure consistency.

Table 3 summarizes the strengths and weaknesses of the various systems discussed above.

Memory compaction: In Carbink, the far memory regions used by a program become fragmented as spans are swapped in. Memory compaction is a well-studied topic in the literature about "moving" garbage collectors for managed languages (e.g., [11, 18, 49]). Moving garbage collection is also possible for C/C++ programs; Mesh [40] represents the state-of-the-art. With respect to this prior work, Carbink's unique challenge is that the compaction algorithm (§3.4.2) must compose well with an erasure coding scheme that governs how objects move between local memory and far memory.

8 Conclusion

Carbink is a far memory system that provides low-latency, low-overhead fault tolerance. Carbink erasure-codes data using a span-centric approach that does not expose swap-in operations to stragglers. Whenever possible, Carbink uses efficient one-sided RMAs to exchange data between compute nodes and memory nodes. Carbink also uses novel compaction techniques to asynchronously defragment far memory. Compared to Hydra, a state-of-the-art fault-tolerant system for far memory, Carbink has 29% lower tail latency and 48% higher application performance, with at most 35% higher memory usage.

Acknowledgments

We thank our shepherd Luís Rodrigues and the anonymous reviewers for their insightful comments. We also thank Kim Keeton and Jeff Mogul for their comments on early drafts of the paper, and Maria Mickens for her comments on a later draft. Yang Zhou and Minlan Yu were supported in part by NSF CNS-1955422 and CNS-1955487.

References

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 106–118, 2020.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, and et al. Remote Regions: A Simple Abstraction for Remote Memory. In *Proceedings of USENIX ATC*, pages 775–787, 2018.

[3] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of ACM HotOS*, pages 120–126, 2019.

[4] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-Rationing Garbage Collection for Hybrid Memories. *ACM SIGPLAN Notices*, 53(4):62–77, 2018.

[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of ACM EuroSys*, pages 1–16, 2020.

[6] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote Memory Calls. In *Proceedings of ACM HotNets*, pages 38–44, 2020.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):53–64, 2012.

[8] Cristina Băsescu and Bryan Ford. Immunizing Systems from Distant Failures by Limiting Lamport Exposure. In *Proceedings of ACM HotNets*, pages 199–205, 2021.

[9] Matthew Burke, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R.K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of USENIX SOSP*, pages 228–242, 2021.

[10] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *ACM Transactions on Storage (TOS)*, 13(3):1–30, 2017.

[11] Jon Coppeard. Compacting Garbage Collection in SpiderMonkey. https://hacks.mozilla.org/2015/07/compacting-garbage-collection-in-spidermonkey/, 2015.

[12] Fernando J. Corbato. A Paging Experiment with the Multics System. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of USENIX NSDI*, pages 401–414, 2014.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[17] Jason Evans. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proceedings of BSDCan Conference*, 2006.

[18] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, 1969.

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of USENIX OSDI*, pages 17–30, 2012.

[20] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of USENIX OSDI*, pages 599–613, 2014.

[21] Google. TCMalloc Open Source. https://github.com/google/tcmalloc.

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of USENIX NSDI*, pages 649–667, 2017.

[23] Xianglong Huang, Stephen M Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The Garbage Collection Advantage: Improving Program Locality. *ACM SIGPLAN Notices*, 39(10):69–80, 2004.

[24] Andrew Hamilton Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond Malloc Efficiency to Fleet Efficiency: A Hugepage-Aware Memory Allocator. In *Proceedings of USENIX OSDI*, pages 257–273, 2021.

[25] Intel. Intel Intelligent Storage Acceleration Library. https://github.com/intel/isa-l.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[27] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Proceedings of Conference on Innovative Data Systems Research*, 2022.

[28] Jakub Łącki, Vahab Mirrokni, and Michał Włodarczyk. Connected Components at Scale via Local Contractions. *arXiv preprint arXiv:1807.10727*, 2018.

[29] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Mitigating the Performance-Efficiency Tradeoff in Resilient Memory Disaggregation. *arXiv preprint arXiv:1910.09727*, 2019.

[30] Jure Leskovec. Friendster Social Network Dataset. https://snap.stanford.edu/data/com-Friendster.html.

[31] Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai. BCStore: Bandwidth-Efficient In-Memory KV-store with Batch Coding. *Proceedings of IEEE International Conference on Massive Storage Systems and Technology*, 2017.

[32] Yuzhe Li, Jiang Zhou, Weiping Wang, and Yong Chen. RE-Store: Reliable and Efficient KV-Store with Erasure Coding and Replication. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 1–12, 2019.

[33] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. Graphlab: A New Framework for Parallel Machine Learning. *arXiv preprint arXiv:1408.2041*, 2014.

[34] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *Proceedings of IEEE International Conference on Big Data*, pages 2884–2892, 2017.

[35] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[36] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Proceedings of Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[37] MongoDB Inc. MongoDB Open Source. https://github.com/mongodb/mongo.

[38] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[39] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, and et al. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[40] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of ACM PLDI*, pages 333–346, 2019.

[41] David M.W. Powers. Applications and Explanations of Zipf's Law. In *Proceedings of New Methods in Language Processing and Computational Natural Language Learning*, 1998.

[42] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of USENIX OSDI*, pages 401–417, 2016.

[43] Irving S. Reed and Gustave Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[44] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of USENIX OSDI*, pages 315–332, 2020.

[45] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[46] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of USENIX OSDI*, pages 69–87, 2018.

[47] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of ACM EuroSys*, pages 1–16, 2020.

[48] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M.K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *Proceedings of ACM SIGCOMM*, pages 93–105, 2021.

[49] SUN Microystems. Memory Management in the Java HotSpot Virtual Machine, 2006.

[50] Michael Sutton, Tal Ben-Nun, and Amnon Barak. Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, pages 12–21, 2018.

[51] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, and et al. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of USENIX OSDI*, pages 787–803, 2020.

[52] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In *Proceedings of ACM EuroSys*, pages 1–14, 2020.

[53] Transaction Processing Performance Council (TPC). TPC-A. http://tpc.org/tpca/default5.asp.

[54] Volt Active Data. VoltDB. https://www.voltdb.com/.

[55] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated Managed Runtime. In *Proceedings of USENIX OSDI*, pages 261–280, 2020.

[56] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid Is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[57] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship Compute or Ship Data? Why Not Both? In *Proceedings of USENIX NSDI*, pages 633–651, 2021.

[58] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does Erasure Coding Have a Role to Play in My Data Center? *Microsoft Research Technical Report*, 2010.

Metastable Failures in the Wild

Lexiang Huang[1,3*], Matthew Magnusson[2*], Abishek Bangalore Muralikrishna[2], Salman Estyak[1], Rebecca Isaacs[3], Abutalib Aghayev[1], Timothy Zhu[1], and Aleksey Charapko[2]

[1] *The Pennsylvania State University,* [2] *University of New Hampshire,* [3] *Twitter*

Abstract

Recently, Bronson et al. [7] introduced a framework for understanding a class of failures in distributed systems called *metastable failures*. The examples of metastable failures presented in that work are simplified versions of failures observed at Facebook. In this work, we study the prevalence of such failures in the wild by scouring over publicly available incident reports from many organizations, ranging from hyperscalers to small companies.

Our main findings are threefold. First, metastable failures are universally observed—we present an in-depth study of 22 metastable failures from 11 different organizations. Second, metastable failures are a recurring pattern in many severe outages—e.g., at least 4 out of 15 major outages in the last decade at Amazon Web Services were caused by metastable failures. Third, we extend the model by Bronson et al. to better reflect the metastable failures seen in the wild by categorizing two types of triggers and two types of amplification mechanisms, which we confirm through developing multiple example applications that reproduce different types of metastable failures in a controlled environment. We believe our work will aid in a deeper understanding of metastable failures and in coming up with solutions to them.

1 Introduction

Building reliable distributed systems has been the holy grail of distributed computing research. Historically, academic researchers studied the reliability of distributed systems under the assumptions of fail-stop [31, 42, 46] and Byzantine [8, 32] failure modes. The proliferation of cloud services led to previously unseen scales and the discovery of new failure modes, such as stragglers [9, 12, 62], fail-slow hardware failures [3, 27, 29], and scalability failures [34, 53]. Most recently, Bronson et al. [7] introduced a new class of failures called *metastable failures*.

Bronson et al. define the *metastable failure state* as the state of a **permanent overload with an ultra-low goodput** (throughput of useful work). In their framework, they also define the *stable state* as the state when a system experiences a low enough load than it can successfully recover from temporary overloads, and the *vulnerable state* as the state when a system experiences a high load, but it can successfully handle that load in the absence of temporary overloads. A system experiences a metastable failure when it is in a vulnerable state and a *trigger* causes a temporary overload that sets off a *sustaining effect*—a work amplification due to a common-case optimization—that tips the system into a metastable failure state. The distinguishing characteristic of a metastable failure is that the sustaining effect keeps the system in the metastable failure state **even after the trigger is removed**.

This phenomenon of metastable failure is not new. However, instances of such failures look so dissimilar that it is hard to spot the commonality. As a result, distributed systems practitioners have given different names to different instances of metastable failures, such as persistent congestion [51], overload [60], cascading failures [5], retry storms [2, 56], death spirals [37], among others. Bronson et al. [7] is the first work that generalizes all of these different-looking failures under the same framework.

A key property of metastable failures is that their root cause is not a specific hardware failure or a software bug. It is an emergent behavior of a system, and it naturally arises from the optimizations for the common case that lead to sustained work amplification. As such, metastable failures are hard to predict, may potentially have catastrophic effects, and incur significant ongoing human engineering costs because automated recovery is difficult (since these failures are not understood well). For example, in Section 6.3, we discuss how code and configuration changes without truly understanding the metastable failure can exacerbate the problem and lead to future incidents. Incidentally, at the time of writing this paper, a metastable failure at Amazon Web Services (AWS) disrupted the operation of airlines [38], home appliances [30], smart homes, payment systems [52], and other critical services for several hours.

As Bronson et al. point out, operators choose to run their systems in the vulnerable state all the time because it is much more efficient than running them in the stable state. As a simple example, an operator of a system with a database that can handle 300 requests per second (RPS) can install a cache with a 90% hit-rate and start serving up to 3,000 RPS. While more efficient, the system is now operating in a vulnerable state because a cache failure can overwhelm the database with more requests that it can handle. The problem is that in a complex, large-scale distributed system, we lack the ability to analyze the consequences of this decision to run in a vulnerable state under different conditions; e.g., what happens if load increases, or if the downstream latency increases, or if messages increase in size and serialization/deserialization starts to cost more CPU? So picking "how vulnerable" of a state to operate in, under normal conditions, is a best guess and not always the right choice, which is why we continue to experience metastable failures.

[*] Equal contribution.

In this paper, we make four contributions that extend the work of Bronson et al. and increase our understanding of metastable failures:

- A study of metastable failures in the wild that confirms metastable failures are universally observed and comprise a substantial fraction of the most severe outages (Section 2).
- An improved model that categorizes two types of triggers and two types of amplification mechanisms, which better explains how metastable failures happen (Section 3).
- An insider view at Twitter of a new type of metastable failure where garbage collection acts as an amplification mechanism (Section 4).
- Three example applications on which metastable failures are experimentally reproduced, which helps researchers propose and test solutions to metastable failures (Section 5). We have open-sourced these examples at https://github.com/lexiangh/Metastability.

We hope our work will encourage more research into this devastating kind of failure and help in building more robust distributed systems, as our daily lives start to increasingly depend on them [20, 30, 38, 52].

2 Metastability in the Wild

Bronson et al. [7] used simplified examples to illustrate the mechanism of metastability and only asserted that the pattern was common, but did not present any data about real-world occurrences. Thus, we perform a large-scale study of actual metastable failures in the wild by sifting through hundreds of publicly available incident reports. It is an arduous task that requires an in-depth analysis of each incident report to understand if the failure is metastable, and the lack of details in the reports makes it even more challenging. We identify 21 metastable failures (Table 1) that are severe enough to warrant public incident reports in a range of organizations, including four at AWS, four at Google Cloud, and four at Microsoft Azure. Though this number may appear low compared to other failure types in distributed systems [26, 27, 33, 53], metastable failures usually have devastating results that last many hours, which makes them an important class of failures to study.

2.1 Methodology

To find examples of metastability, we searched through troves of publicly available post-mortem incident reports from large cloud infrastructure providers and significantly smaller companies or services. Large infrastructure providers, such as Amazon Web Services (AWS), Azure, and Google, are held accountable by many paying customers, forcing greater transparency into their reliability and operation practices. Smaller businesses often operate with higher self-imposed transparency goals until they grow large enough to become a significant target for malicious attacks.

Infrastructure providers often maintain incident and outage reporting tools [4, 11, 50], which became our primary source for metastable failures. We analyzed hundreds of incidents to find a handful that depicts systems in the metastable state. We also found several smaller failures from other public sources such as postmortem communities [13, 44, 45, 54], weekly outage incident digests [14, 17, 55], etc.

The reports from different sources do not follow the same format nor provide the same level of information, making our job of finding examples of metastability more difficult. While going through these reports, we focus on tell-tale signs of metastability—temporary triggers, work amplification or sustaining effects, and certain specific mitigation practices. More specifically, we look for patterns when a trigger initiates some processes that amplify the initial trigger-induced problem and sustain the degraded performance state even after the trigger is removed. The sustaining effect can take multiple forms, such as exacerbated queue growth or retries that create more load. We also pay attention to mitigation efforts, as metastable failures often require significant load shedding [57, 60] for recovery.

We perform a comprehensive analysis of these incidents, focusing on impact, trigger, work amplification mechanisms, and mitigation practices. To study the impact, we focus on the duration and number of impacted services. This information is usually readily available in the reports. For the triggers, we identify the triggers and classify them into several distinct categories. We use a similar identification and classification process to distill work-amplification mechanisms and mitigation patterns. We present our summarized findings in Table 1.

2.2 Summary of Metastable Failures in the Wild

In Table 1, we provide a breakdown of metastable failure incidents we have found. The examples include instances from both major cloud providers (e.g., Microsoft, Amazon, Google, IBM) and smaller companies and projects (e.g., Spotify, Elasticsearch, Apache Cassandra). Our summary table describes high-level aspects of these failures: duration of the incident, impacted services, triggers leading to the outage, the sustaining effect mechanism, and corrective actions taken by the engineers.

Due to the often limited scope of provided information, we use our best judgment in identifying metastable failures. The most important criteria we use is the sustaining effect mechanism. We highlight several instances in gray color when the incident description is not clear on the presence of such a sustaining effect, but metastable failure is plausible depending on the interpretation and given the rest of the information provided. Additionally, we assign each incident a unique identifier to refer to each incident later.

Triggers are the starting events in the chain leading to metastable failures. Around 45% of observed triggers in Table 1 are due to engineer errors, such as buggy configuration or code deployments, and latent bugs (i.e., undetected pre-existing bugs). These can be observed in incidents GGL1, GGL2, GGL3, GGL4, AWS1, AWS3, AZR3, ELC1, SPF1. Load spikes are another prominent trigger category, with around 35% of incidents reporting it. A significant number of cases (45%) have more than one trigger.

	ID	Date	Duration (hours)	Services Impacted	Triggers	Sustaining Effect	Mitigation
Google	GGL1 [22]	03/12/19	4.17	Gmail, Photos, Drive, Cloud Storage, various other GCP services	• load spike • config change	• cascading overload	• load shedding • stop config deploy
	GGL2 [23]	10/31/19	21.5	multiple components of GCE	• software bug	• retry	• load shedding • reboot • capacity increase
	GGL3 [24]	04/08/20	3.2	Google BigQuery, Cloud IAM 3% of Cloud SQL HA	• config change • software bug	• retry	• config rollback • policy change
	GGL4 [21]	04/30/13	1.5	Google API infrastructure	• config change • latent software bug	• traffic queue growth • reboots	• config rollabck • server reboot
AWS	AWS1 [47]	04/21/11	66.7	Amazon EC2, Amazon RDS	• network config change	• retry	• config rollback • policy change • load shedding • capacity increase
	AWS2 [48]	06/13/14	4.23	Amazon SimpleDB	• power loss	• retry	• load shedding • server restart
	AWS3 [49]	09/20/15	4.55	AWS SQS, EC2 Autoscaling, CloudWatch, AWS Console	• load spike • network disruption	• retry • cascading server demotion	• load shedding – pause metadata ops • capacity increase
	AWS4 [51]	12/07/21	9.3	AWS DynamoDB, EC2, Fargate, RDS, EMR, Workspaces, AWS Console, Authorization services, internal DNS	• latent software bug triggered by scale-up led to load spike	• retry	• load rebalancing • load shedding
Azure	AZR1 [4]	07/01/20	2.65	Azure SQL DB & SQL Data Warehouse, Azure Database for MySQL/PostgreSQL/MariaDB	• unspecified load imbalance trigger • latent config bug	• cascading overload	• service restart
	AZR2 [4]	04/01/21	1.15	Azure DNS	• software bug leading to cache degradation	• retry	• unknown automation • capacity increase
	AZR3 [4]	06/14/21	13.25	Management operations of many Azure Services	• latent software bug • load spike	• unspecified queue growth due to overload and timeouts	• load shedding • remove buggy software • capacity increase
	AZR4 [4]	07/12/21	7.92	Windows Virtual Desktop, Azure Front Door, Azure CDN Standard	• deployment of software bug • load spike	• retry • other unspecified	• load rebalancing • trigger hot fix • policy change
Other	IBM1 [11]	06/11/21	73.53	Private DNS, HS Crypto Service, Cloudant DNS Services, Osaka, Cloudshell services	• software bug	• retry	• load shedding • policy change • trigger hot fix
	SPF1 [19]	04/13	NA	core app/service UI	• load spike • policy failure	• retry	• load shedding
	SPF2 [19]	06/04/13	8.33	core app/service UI	• load spike due to unexpected service dependency	• retry • excessive logging in failure case	• trigger hot fix • load shedding
	ELC1 [39]	04/02/19	6.67	Elasticsearch Service	• unspecified maintenance • unspecified error	• load caused ZK churn causing more load	• restart • load shedding
	WIK1 [58]	03/30/21	2.25	media upload, misc queued jobs	• load spike	• unspecified causing queue growth	• load shedding • policy change
	CCI1 [10]	07/07/15	18.33	Core product	• load spike	• load increase due to contention	• load shedding
	CAS1 [1]	07/27/17	NA	Partial database outage	• rolling restart	• self-sustaining and increasing overload	• policy change
	CAS2 [43]	2020	0.16	ably services	• load spike of certain costly operations	• retry	• trigger removal – operated in stable state
	FB1 [18]	NA	NA	Facebook core services	• load spike	• software bug	• hot fix

Table 1: Metastable failures from public sources. Azure and IBM do not provide a direct incident link. Gray highlight indicates a plausible metastable failure, although the incident description lacked some necessary details.

Handling and recovering from metastable failures is not easy, with our data suggesting that incidents cause significant outages. For instance, the IBM1 incident lasted over three days. More generally, we have observed outages in a range of 1.5 to 73.53 hours, with 4 to 10 hours of outages being the most common (35% of incidents reporting the outage period).

While triggers initiate the failure, the sustaining effect mechanisms prevent the system from recovering. We observed a variety of different sustaining effects, such as load increase due to retries, expensive error handling, lock contention, or performance degradation due to leader election churn. By far, the most common sustaining effect is due to the retry policy, affecting more than 50% of the studied incidents—GGL2, GGL3, AWS1, AWS2, AWS3, AZR2, AZR4, IBM1, SPF1, SPF2, and CAS2 incidents are all sustained by retries.

Recovery from a metastable failure is challenging and often requires reducing load. Direct load shedding, such as throttling, dropping requests, or changing workload parameters,

Symbols	Names
L_{norm}, C_{norm}	Normal load and capacity without issues (i.e., triggers)
$L_{org}(t), C_{org}(t)$	Organic load and capacity at time t including effects from triggers
$L_{sys}(t), C_{sys}(t)$	System load and capacity at time t including metastable amplification over the organic load and capacity
C_{stable}	Stable capacity below which the system recovers from metastability
m_{trigL}, m_{trigC}	Maximum load-spike and capacity-decreasing trigger magnitudes
$\alpha_L(t)$, $\alpha_C(t)$	Workload and capacity degradation amplification factors
Δt_{trig}	Trigger overloading duration
$w_L(\Delta t_{trig})$, $w_C(\Delta t_{trig})$	Workload and capacity degradation amplification upper bound functions
w_L^*, w_C^*	Maximum workload and capacity degradation amplifications

Table 2: Symbols of Metastability Framework.

was used in over 55% of the cases. Some indirect mechanisms were also popular, such as reboots to clean the queues or operation backlogs, or policy changes. An example of such a policy change is the CAS1 incident where a feature was turned off to allow the servers to join the cluster.

3 Metastability Framework

Based on our observations of real-world metastable failures, we extend the model of Bronson et al. [7] in three ways. First, while the previous framework presumes that a system entering a metastable failure state is usually due to a load increase, we observe in multiple incidents that a software bug or a configuration change may decrease the capacity of the system and trigger a metastable failure even without a load increase. Second, although the previous framework describes a system sustaining in a metastable failure state due to workload amplification, we show examples of another type of metastable failure sustaining effect where background activities such as garbage collection cause the system's capacity to degrade or remain degraded even after the trigger is removed. Third, based on our experiments on the reproductions of metastable failures, we find that a vulnerable state is not a binary condition; whether a system transitions from a vulnerable state into a metastable failure state is determined by the current degree of vulnerability, the trigger magnitude, and its duration.

3.1 System Model

We devise our model based on the load and capacity of a system, and a summary of the symbols are shown in Table 2. The capacity of the system, $C_{sys}(t)$, is represented in terms of abstract resource units (RUs) that the system can handle per second (i.e., work per second). Each request consumes some RUs from the system's budget. For example, consider a system with a constant $C_{sys}(t) = 100$ RUs/sec; every second such a system can process up to 100 requests, each costing 1 RU, or up to 50 requests, each costing 2 RUs. The load, $L_{sys}(t)$, represents the work per second arriving to the system in terms of RUs/sec. So for a system to not be overloaded, $L_{sys}(t) < C_{sys}(t)$.

Under normal idealized conditions, we assume the processing capacity $C_{sys}(t)$ is constant, $C_{sys}(t) = C_{norm}$. However, depending on circumstances it may diminish due to failures, transient outages, or amplification effects of metastability. Similar to $C_{sys}(t)$, we set $L_{sys}(t) = L_{norm}$ as the normal load excluding transient effects and workload amplification.

Since metastability is fundamentally due to sustaining effects that amplify the load and degrade the system capacity, we also define $L_{org}(t)$ and $C_{org}(t)$ as the load and system capacity without amplification effects. That is, the organic load, $L_{org}(t)$, is the load originating from the system's clients. This includes transient effects such as load spikes, but does not include workload amplification effects such as retries. Similarly, the organic capacity, $C_{org}(t)$, represents the system capacity including transient capacity decreases, but without sustaining degradation. For example, background interference may drop the organic capacity in half temporarily until the interference ends. But sustaining amplification effects such as garbage collection would cause the system capacity to degrade further or remain degraded even after the trigger is removed. We illustrate these effects on capacity and load in Figure 1.

3.2 Triggers

Metastable failures begin with trigger events. In our survey (Section 2), we have identified two broad types of triggers. The first trigger type results from a sudden burst in organic load, $L_{org}(t)$ (e.g., a celebrity posting their baby's picture). The left two scenarios in Figure 1 illustrate how such a trigger could lead to a metastable failure, and incidents GGL1, AWS3, AZR3, AZR4, SPF1, SPF2, WIKI1, CCI1, and CAS2 are examples of such failures. The second trigger type degrades the system's organic capacity, $C_{org}(t)$ (e.g., a rack failure or deployment of inefficient code). The right two scenarios in Figure 1 illustrate how such a trigger could lead to a metastable failure, and incidents GGL2, GGL3, GGL3, AWS1, AWS2, AZR4, and IBM1 are examples of such failures. While the two types of triggers behave differently, they impact the system's operation similarly by changing the balance between the load and capacity.

Definition 1 (Trigger). A **trigger** $\mathcal{T}(m_{trigL}, m_{trigC})$ represents the total effect from one or more of the following events:

- A **load-spike trigger** is an event that increases the load on the system by some maximum magnitude m_{trigL} such that $L_{org}(t) - L_{norm} \leq m_{trigL}$ for all t.
- A **capacity-decreasing trigger** is an event that decreases the system capacity by some maximum magnitude m_{trigC} such that $C_{norm} - C_{org}(t) \leq m_{trigC}$ for all t.

We assume m_{trigL} and m_{trigC} represent upper bounds on the total trigger effect across all the triggers in a trigger event. In our survey, over half of the observed incidents had one

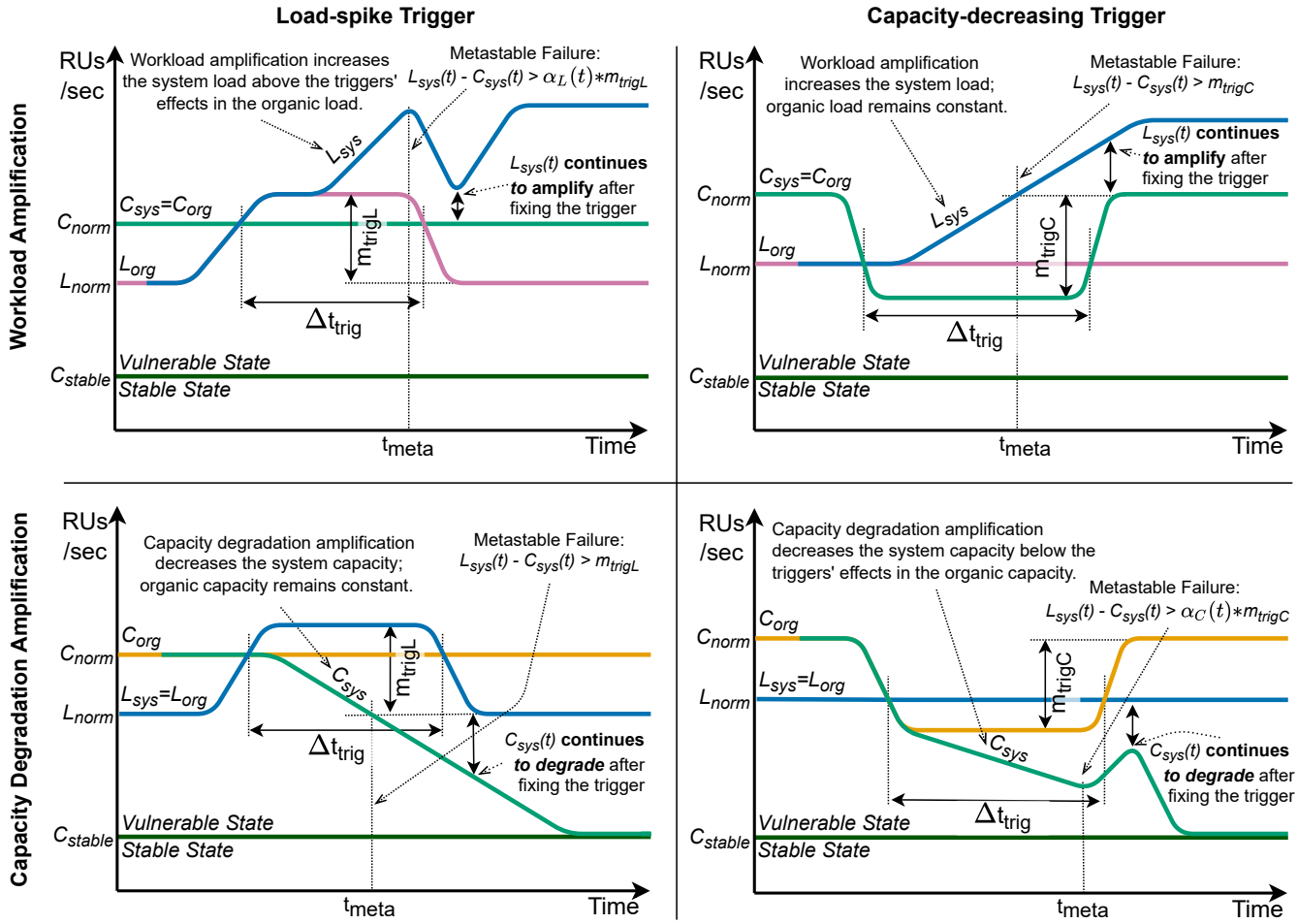


Figure 1: Four metastability scenarios (Section 3.4). Two types of triggers (i.e., load-spike and capacity-decreasing) and two types of amplification mechanisms (i.e., workload amplification and capacity degradation amplification) form the different scenarios.

overloading trigger, but to our surprise, incidents with multiple triggers were also common (GGL1, GGL3, GGL4, AWS3, AZR1, AZR3, AZR4, SPF1, and ELC1).

Not all triggers are dangerous; small variations of capacity or load are normal and unavoidable. The triggers become dangerous when they overload the system (i.e., $L_{sys}(t) \geq C_{sys}(t)$).

Definition 2 (Overloading trigger condition). If $m_{trigL} + m_{trigC} \geq C_{norm} - L_{norm}$, then the trigger(s) can overload the system.

Theorem 1 (Overloading trigger). If the system does not have an overloading trigger condition, then it will never have a metastable failure. (Proof in Section A.1.)

An overloading trigger is a necessary precursor of a metastable failure. Once the system is in an overloaded state, its performance starts to degrade, which sets off alarms and starts mitigation efforts. For instance, GGL2, GGL3, AZR2, and AZR4, among others, relied on an automated monitoring and notification system to initiate the mitigation efforts once a drop in availability is detected. Although not always explicitly mentioned, we believe that most, if not all, systems surveyed have monitoring and notification capabilities. As the system overloads, the latency of client operations will start to rise, while the goodput will stay at or below the $C_{sys}(t)$.

The overloaded state, however, is not a metastable failure state just yet. Getting out of overload is relatively straightforward — fix the trigger and restore the balance in the system, such that $L_{sys}(t) < C_{sys}(t)$ again. If the load on the system returns to a level below the system's capacity when the trigger is removed, then the system should eventually recover.

The duration of the trigger's impact on the system is another important aspect to consider. While the trigger persists, the system is working in a reduced capacity or increased load setting, depending on the trigger type. Some triggers are easy to fix and end their impact on the systems. For instance, most of the misconfiguration triggers (GGL1, GGL4, AWS1) and buggy software deployment (AZR4) can be fixed by a rollback action. Some triggers are more difficult to address since the ability to fix the trigger, ironically, may depend on the system's performance, which is degraded by the trigger. For instance, CAS1 incident had a rolling server restart for maintenance. The restart lowers the overall capacity of the system by the capacity of a server currently rebooting, placing more

load on other machines. A rebooted server needs to repair itself by catching up, which requires both the rebooted server to be fast enough to catch up with the missed and ongoing load, and the rest of the system needs to be fast enough to provide this repair service.

3.3 Sustaining Effect Loop

A metastable failure arises when the overloaded system does not eventually return to a healthy state ($L_{sys}(t) < C_{sys}(t)$). Many reasons can keep the system in a failed state, such as the inability to recover from the trigger or an uncontrolled increase in load, among others. However, all of the reasons share a common pattern of keeping systems inoperative. We refer to this pattern as a *sustaining effect*.

Definition 3 (Sustaining effect). A **sustaining effect** is a feedback loop that keeps the system in an overloaded state such that $L_{sys}(t) \geq C_{sys}(t)$ even after the trigger is removed.

The feedback mechanism itself may have existed prior to the trigger, however, the overload made the feedback mechanism self-sustaining, and we name this feedback mechanism as *metastable amplification*. For instance, the AWS4 incident occurred with networking overload due to a planned scale-up operation. The overload resulted in connection timeouts and retries, creating even more load and causing more timeouts and retries.

Definition 4 (Metastable amplification). A **metastable amplification** $\mathcal{W}(\alpha_L(t), \alpha_C(t), w_L(\Delta t_{trig}), w_C(\Delta t_{trig}), w_L^*, w_C^*)$ exacerbates the system's overload until it reaches a maximum overload limit. The amplification can manifest itself by increasing the load on the system $L_{sys}(t)$ and/or decreasing the system's capacity $C_{sys}(t)$:

- **Workload amplification** is a feedback loop that increases the system load $L_{sys}(t)$ beyond the organic load $L_{org}(t)$ (i.e., $L_{sys}(t) \geq L_{org}(t)$). The workload amplification factor, $\alpha_L(t) = L_{sys}(t)/L_{org}(t)$, can be upper bounded by some workload amplification upper bound function $w_L(\Delta t_{trig})$ and max load amplification w_L^* such that $1 \leq \alpha_L(t) \leq w_L(\Delta t_{trig}) \leq w_L^*$ for all $t, \Delta t_{trig}$, where w_L is a monotonically increasing function of the trigger overloading duration Δt_{trig} from $w_L(0) = 1$ to $w_L(\infty) = w_L^*$.
- **Capacity degradation amplification** is a feedback loop that decreases the system's capacity $C_{sys}(t)$ below the organic capacity $C_{org}(t)$ (i.e., $C_{sys}(t) \leq C_{org}(t)$. The capacity degradation amplification factor, $\alpha_C(t) = C_{sys}(t)/C_{org}(t)$, can be upper bounded by some capacity degradation amplification upper bound function $w_C(\Delta t_{trig})$ and max capacity degradation amplification w_C^* such that $1 \geq \alpha_C(t) \geq 1/w_C(\Delta t_{trig}) \geq 1/w_C^*$ for all $t, \Delta t_{trig}$, where w_C is a monotonically increasing function of the trigger overloading duration Δt_{trig} from $w_C(0) = 1$ to $w_C(\infty) = w_C^*$.

Intuitively, the upper bounds allow us to reason about vulnerability and when a system enters a metastable failure state. We do not assume the upper bounds are tight, and the intent is to explain (i) there are two different types of amplification (that may both be active simultaneously), and (ii) how the amplification factors impact metastability.

Workload amplification can manifest in multiple ways. Recall that each request in our model has some RU cost. The workload amplification, therefore, can use one of the two broad mechanisms—increasing the number of requests in the system or increasing the average cost of a request. We observe the former amplification method in incidents GGL2, SPF1, AZR2, while the latter shows up in AWS3, WIKI1, and SPF2. For example, the SPF1 incident was caused by retrying the requests, while the SPF2 issue was exaggerated by extra debug logging added for timed-out requests. In our model, this corresponds to the top left scenario in Figure 1 where a load-spike trigger (i.e., L_{org} increases) starts a workload amplification (i.e., L_{sys} increases) due to retries or an increase in the average per-request cost.

Workload amplification does not necessarily start immediately with the trigger. A common type of workload amplification is retry-driven amplification, observed in incidents GGL2, AWS1, AWS2, AZR2, SPF1, and SPF2. It occurs when the requests start to timeout after waiting for some timeout period, and clients begin to retry the failed requests. Typically, this type of amplification starts building after some amplification delay. This delay depends on several factors, such as the degree of overload in the system and request timeout. A short request timeout is good for latency when retrying due to a small transient issue. However, it can hurt the system's ability to handle larger problems by quickly starting the workload amplification. For example, AWS2 specifies that a small handshake timeout was a contributing factor to starting and sustaining the overload. The handshake timeout controlled the frequency of heartbeat messages and the duration a server can remain active without receiving a heartbeat. A longer timeout would have both reduced the heartbeat load and allowed for a longer heartbeat wait, potentially delaying workload amplification.

Capacity degradation amplification is another common type of sustaining effect. This effect occurs when the initial trigger overloads the system and causes the capacity to degrade or remain degraded. For example, a system experiencing a trigger where background interference from other co-located processes pushes it into an overloaded state may now need to also deal with an increased amount of garbage collection (GC) due to a queue buildup. In this case, the GC amplifying effect would degrade the system capacity beyond the capacity decrease from the trigger. The metastability arises when the capacity degradation from GC grows to be high enough such that the system remains overloaded even if the background interference is removed.

A sustained degradation is a special case of capacity degradation amplification. The CAS1 incident discussed earlier is an example of this. A different instance of this type of sustaining effect is the caching failure described in Bronson et al. [7]. In a system backed by a look-aside cache, a partial failure of a cache, such as a reboot of a caching server, may

result in a load spike on the underlying database and cause it to timeout. Because of timeouts, all operations effectively fail, preventing the system from filling the cache. This sustaining effect causes the capacity to remain degraded.

Figure 1 shows the impact of workload amplification (top scenarios) and capacity degradation amplification (bottom scenarios) on metastability. While the scenarios look different visually, they can all be understood under the metastability model of a sustaining effect that amplifies the load and/or capacity degradation to magnify an overload condition.

3.4 Metastability Scenarios

Figure 1 demonstrates four scenarios in which metastable failures occur. We introduce two types of triggers and two types of amplification mechanisms that impact the load and capacity of the system. We use the terms "load-spike" for triggers and "workload amplification" for amplification mechanisms impacting the load, and we use the terms "capacity-decreasing" for triggers and "capacity degradation" for amplification mechanisms impacting the capacity. In practice, both types of triggers and both types of amplification mechanisms can occur simultaneously.

In the upper left scenario, when there is a load-spike trigger, the organic load L_{org} increases beyond the system capacity C_{sys}, and thus the system is overloaded. The workload amplification (e.g., retries) further increases the system load L_{sys} above the triggers' effects (i.e., m_{trigL}) in the organic load L_{org}. When the system overload (i.e., $L_{sys}(t) - C_{sys}(t)$) is high enough (e.g., at time t_{meta}), even after removing the trigger (i.e., the dip of L_{org} and L_{sys}), the system remains overloaded and the workload amplification mechanism continues to exacerbate the overload, which indicates the system is in a metastable failure state.

In the upper right scenario, when there is a capacity-decreasing trigger, the organic capacity C_{org} decreases below the system load L_{sys}, and thus the system is overloaded. The workload amplification (e.g., retries) increases the system load L_{sys}. Once the amplification is high enough (e.g., at time t_{meta}), even after removing the trigger (i.e., the recovery of C_{org} by m_{trigC}), the system is still overloaded, and the workload amplification mechanism continues to exacerbate the system overload. Hence a metastable failure.

In the bottom left scenario, when there is a load-spike trigger, the organic load L_{org} increases beyond the system capacity C_{sys}, and thus the system is overloaded. The capacity degradation amplification (e.g., GC amplifying effect) decreases the system capacity C_{sys}. Once the amplification is high enough (e.g., at time t_{meta}), even after removing the trigger (i.e., the organic load L_{org} decreases by m_{trigL}), the system is still overloaded, and the capacity degradation amplification mechanism continues to exacerbate the system overload. Hence a metastable failure.

In the bottom right scenario, when there is a capacity-decreasing trigger, the organic capacity C_{org} decreases below the system load L_{sys}, and thus the system is overloaded. The capacity degradation amplification (e.g., GC amplifying effect) further decreases the system capacity C_{sys} below the triggers' effects (i.e., m_{trigC}) in the organic capacity C_{org}. When the system overload (i.e., $L_{sys}(t) - C_{sys}(t)$) is high enough (e.g., at time t_{meta}), even after removing the trigger (i.e., the recovery of C_{org} and C_{sys}), the system remains overloaded and the workload amplification mechanism continues to exacerbate the overload. Hence a metastable failure.

3.5 System States

Based on Bronson et al. [7], we define three states (stable, vulnerable, metastable failure) that a system operates in and describe the boundaries between these states.

3.5.1 Stable State

Assuming a system has a metastable amplification mechanism $\mathcal{W}(\alpha_L(t), \alpha_C(t), w_L(\Delta t_{trig}), w_C(\Delta t_{trig}), w_L^*, w_C^*)$ and trigger $\mathcal{T}(m_{trigL}, m_{trigC})$, it will never have a metastable failure if it's running under low enough load $L_{norm} < C_{stable}$. The demarcation line between stable and vulnerable states depends on the max amplification factors w_L^*, w_C^* and the normal capacity of the system, C_{norm}.

Theorem 2 (Stable region). Define $C_{stable} = \frac{C_{norm}}{(w_L^* * w_C^*)}$. If $L_{norm} < C_{stable}$, then the system will never have a metastable failure. (Proof in Section A.2.)

When the normal load is low enough relative to the normal system capacity, then even if the trigger overloads the system and causes the maximum metastable amplification, it will recover once the trigger is removed and hence is not a metastable failure. For instance, in the CAS2 incident, the Apache Cassandra cluster operated at a low load of 10% to 30% percent of the capacity. Despite a very significant trigger and workload amplification, the cluster recovered itself when the trigger was removed.

3.5.2 Vulnerable State

If the system has a normal load higher than C_{stable}, it's running in a vulnerable state. Bronson et al. [7] define the vulnerable state as the state when a system experiences a high enough load that temporary overloads can tip the system into a metastable failure state. However, based on our experiments, the vulnerable state is not a binary—there are many degrees to it and many factors determine this degree of vulnerability.

As an overloading trigger event $\mathcal{T}(m_{trigL}, m_{trigC})$ unfolds, the system (and engineers) are in a race to mitigate the overload before the feedback loop of the sustaining effect makes the failure unrecoverable without more drastic measures. In such a system, a combination of amplification and trigger factors impact the likelihood of a metastable failure.

Theorem 3 (Degrees of vulnerability). If the metastable amplification during the trigger overloading duration Δt_{trig} is small enough relative to the system headroom (i.e., $w_L(\Delta t_{trig}) * w_C(\Delta t_{trig}) < \frac{C_{norm}}{L_{norm}}$), then the system will never have a metastable failure. (Proof in Section A.3.)

Once the system is in a vulnerable state, a combination of factors determines its degree of vulnerability. First, how close L_{norm} is to C_{norm} impacts the vulnerability. The smaller the

C_{norm}/L_{norm} ratio, the easier it is to enter a metastable failure state (i.e., the higher degree of vulnerability). The smaller the $C_{norm} - L_{norm}$ difference, the smaller the trigger magnitude needed to overload the system and potentially trigger the metastable failure (Theorem 1). Second, the metastable amplification impacts the vulnerability. As described in Theorem 3, higher metastable amplifications ($w_L(\Delta t_{trig})$ and $w_C(\Delta t_{trig})$) increase the vulnerability to metastable failures. Since w_L and w_C increase with the overloading trigger duration Δt_{trig}, longer triggers also increase the vulnerability.

The amplification delay, if present, is the first mechanism to buy some time for mitigation efforts. Unfortunately, there is very little control over this delay interval, aside from timeouts in scenarios like retry-based workload amplification. The trigger overloading interval Δt_{trig} is another factor in determining whether an overload develops into the metastable failure. Intuitively, short triggers mean that amplification may not have started yet due to the amplification delay or has not escalated too far. Recall that entering a metastable failure state requires the system load to exceed the capacity even after fixing the trigger. This means that the amplification factors $\alpha_L(t)$ and $\alpha_C(t)$ play a role—a smaller amplification translates into a more moderate system load growth that can buy the engineers time to recover the trigger.

3.5.3 Metastable Failure State

The point when the trigger(s) cause the system to enter a metastable failure state depends on the current amplification factors $\alpha_L(t)$ and $\alpha_C(t)$ and trigger magnitudes.

Theorem 4 (Metastable failure boundary). If the metastable amplification causes the system overload to exceed the triggers' effects (i.e., $L_{sys}(t) - C_{sys}(t) \geq \alpha_L(t) * m_{trigL} + \alpha_C(t) * m_{trigC}$), then the system is in a metastable failure state. (Proof in Section A.4.)

Since the current amplification factors and trigger magnitudes change over time, we can use the current amplification factors and maximum trigger magnitudes to develop a metastable failure boundary. If the overload $L_{sys}(t) - C_{sys}(t)$ exceeds the boundary in Theorem 4, then there is a metastable failure because the system is overloaded even after the trigger is removed. If the overload $L_{sys}(t) - C_{sys}(t)$ is below the boundary in Theorem 4 while the trigger(s) are in full effect, then the system is not in a metastable failure state yet. This is because the removal of the trigger would result in a non-overloaded state where the system can recover.

Theorem 4 indicates the boundary in the general case where both types of triggers and amplifications occur simultaneously, but for simplicity, Figure 1 depicts the specific boundaries for each type of trigger and amplification in the four scenarios. That is, $m_{trigL} = 0$ or $m_{trigC} = 0$ depending on the trigger, and $\alpha_L(t) = 1$ or $\alpha_C(t) = 1$ depending on the amplification.

A practical takeaway from these results is that it is important to monitor the overload and take more drastic measures before it exceeds the metastable failure boundary. The key insight is that the overload should not be so bad that the system is overloaded even after the trigger is removed.

3.6 Recovery

Fixing the trigger is the first intuitive step many engineers take in recovery efforts. The intuition is likely the result of treating the trigger as the root cause of the failure. For instance, many incidents caused by deploying bad configuration involved rollbacks (GGL3, GGL4, AWS) or halting the deployments (GGL1). Similarly, many incidents triggered by software bugs involved hot-fixing the bug (AZR4, IBM1, SPF2). All incidents caused by load spikes included some form of load shedding (GGL1, AWS3, AZR3, CCI1, etc.).

However, once in the metastable failure state, the system cannot recover all by itself as the sustaining effect keeps it in the metastable failure state. Therefore, we need to remove the sustaining effect from the system to recover. Two broad strategies exist to recover from the failure. The first is load shedding—bringing the load down below the stable threshold C_{stable}. The second is to raise C_{stable} by increasing the system capacity.

Load shedding was the most popular mitigation effort used in over 50% of the incidents. This approach is intuitive in any kind of overload situation. However, without a proper understanding of the metastability and feedback loops, it is hard to know just how much the load needs to be reduced. This results in long mitigations and additional destructive steps, such as server reboots (AWS2, GGL2).

Raising C_{stable} is more nuanced than load shedding. One mechanism for changing the stable threshold is a policy change that impacts the amplification thresholds w_L^* and w_C^*. An example of such a policy change is decreasing the maximum number of retries per request. For instance, a policy with at most two retries will not amplify the work more than three times, while the policy with no cap effectively leaves the system with no stable region. A more popular way of increasing C_{stable} is to add the capacity to the system, essentially raising its C_{norm}. For a fixed w_L^* and w_C^*, increasing normal capacity will also raise the stable threshold, per Theorem 2. A few incidents in our study used this approach. For instance, AZR2 added more capacity after performing load shedding and fixing the trigger.

4 Metastability at Twitter

While publicly available incident reports provide enough high-level information to identify the metastable failures, they lack the depth and detail to understand the complex interactions between components in large systems. In this case study, we use insider information to describe in detail one specific metastable failure occurring at Twitter, a large internet company, due to garbage collection (GC). We identify a sustaining loop where high queueing increases memory pressure and mark-and-sweep processing during GC, causing job slowdowns and thus higher queueing. The effect is more pronounced at high system loads, where the system is more vulnerable to spikes. Specifically, we see that a peak load test

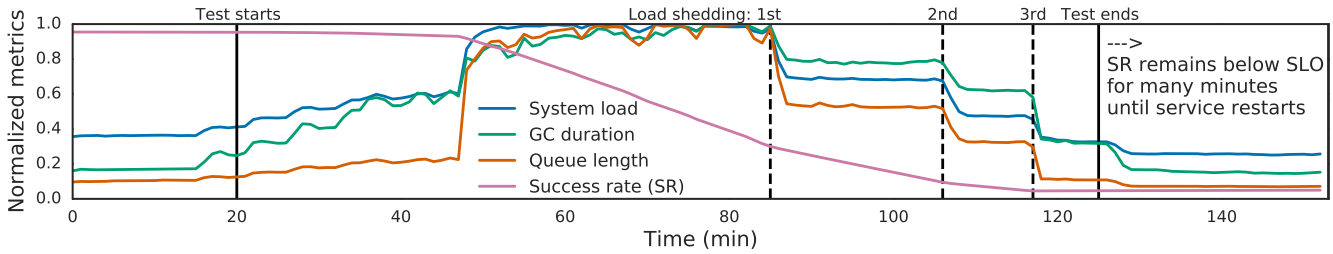
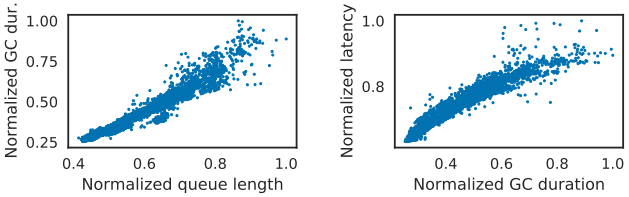


Figure 2: Timeseries of a core service under a peak load test at Twitter. Metrics are normalized except for the success rate, which is scaled to show the trend dropping below the SLO.



(a) Queue length vs. GC duration. (b) GC duration vs. Latency.

Figure 3: Correlation between metrics during 3 normal days.

during a busy day triggers the system to enter a metastable failure state where jobs start to fail, and it is only after sufficient load shedding that the success rate stops dropping.

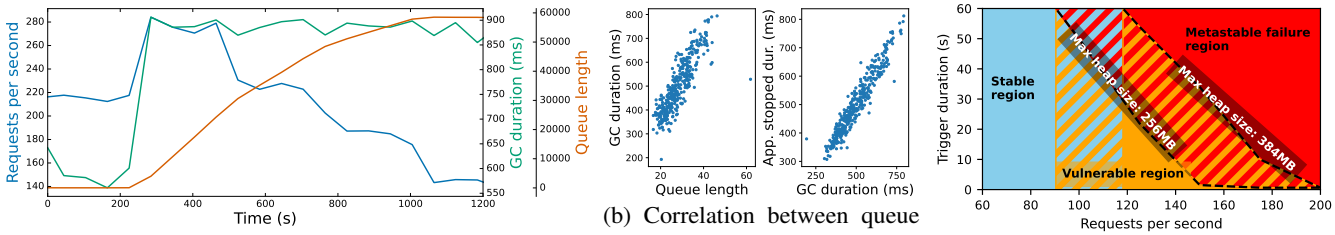
Peak load tests are one of the common types of tests used regularly in industry to expose potential problems and highlight the necessary steps to prevent incidents from happening. Figure 2 shows the timeseries of system metrics at a core service during a peak load test where we see a metastable failure. System load, GC duration, and queue length have been normalized to show the trend only, while success rate (SR) is scaled to demonstrate it dropping sharply below the SLO. All metrics are measured using the standard observability tools at Twitter, except for the (average) queue length, which is inferred using Little's Law [35]. By queue length, we mean the count of all the requests in the system. The service is a mature production service that's well-tuned and has been running for several years, under all the usual operating practices of frequent deployments, regular stress tests, and continuous monitoring and alerting.

In this incident, the peak load occurs around the 48-minute mark, and the SR starts to drop over time. Once the SR of this service drops below a critical threshold (i.e., the SLO), service operators are alerted to mitigate the problem. In this incident, the operators start load shedding at around the 83-minute mark and continue with more load shedding at 106 minutes. This had the desired effect of lowering the load, which also lowers GC and queue length. However, the SR still continues to drop and does not start to recover even when the load is back down to the level before the test. SR remains below the SLO until the service is restarted by operators. This is because even after the load shedding, a sustaining effect is still slowing down the system and causing it to remain in a metastable failure state.

Studying the internal system metrics from the test has shed some light on the problem. We find that the changes to GC duration are highly correlated with load fluctuations, as more load brings more memory allocation, thus requiring more GC. However, the GC is busier than normal during the peak load test. During the second load-shedding period between 106-118 minute marks, the load is more than 20% lower than that at the 40-minute offset, yet the GC is busier and SR is still dropping. At the same time, the queue length is also more than 50% higher, which implies that there are more jobs stuck in the system exacerbating GC. Thus, there is contention between arriving traffic and GC consuming resources, suggesting the metastability sustaining effect.

Specifically, the incident is caused by the sustaining effect in the following steps: (i) a load spike (i.e., a L_{org} increase) caused by peak load test introduces initial high queue length in the system; (ii) high queue length results in high GC behaviors; (iii) high GC behaviors slow job processing (i.e., C_{sys} decreases); (iv) more jobs get stuck in the system, which leads to higher queue length.

To demonstrate each of these steps, we further study data from this test as well as non-test data as a baseline. For (i), we can see the initial trigger in Figure 2 at around minute 48 where the load spike causes a sharp increase in queue length. For (ii), we see that queue length and GC duration are correlated over time in Figure 2. Additionally, we plot queue length vs. GC duration (Figure 3a) under 3 normal days without the test to show these metrics generally exhibit a positive correlation. One might wonder whether the system load affects these metrics, and we find that it is correlated to both queue length and GC duration. But to eliminate the impact of system load, we also filtered the data to only include results with approximately the same system load, and we still see a correlation between queue length and GC duration, which suggests that high queue length leads to high GC. Correlation does not imply causation, so we validate and reproduce these effects in Section 5.1 via a simple example. For (iii), we plot GC duration vs. latency (Figure 3b) during the same period without peak load testing and observe that the latency increases with GC duration. As GC consumes CPU cycles, there is CPU contention with job processing, which causes slowdowns to jobs as evidenced by the higher latencies. Naturally, job slowdowns will cause additional congestion and queueing, which completes the sustaining loop (iv).



(a) Timeseries of GC metastability.

(b) Correlation between queue length, GC, & application stalls.

(c) Degrees of vulnerabilities.

Figure 4: Metastability in Garbage Collection (GC).

Similar incidents recur many times, and engineers take different approaches to mitigate/fix this issue. For example, (i) observing unusually high latency spikes in backend services resulted in work to improve their performance to lower queue lengths, (ii) observing higher GC duration than normal resulted in adjusting the JVM memory configuration (e.g., increasing max heap size) to tweak GC behavior, and (iii) observing high resource utilization (e.g., CPU) resulted in adding more servers to lower per-server load. These approaches decrease system vulnerabilities and make it more robust to the trigger at the magnitude of the peak load test level.

5 Replicating Metastability

We introduce three example applications and experimentally reproduce metastable failures on them. One of these applications reproduces the failure in the Twitter case study (Section 4) at a small scale, and the other two reproduce failures due to retries and look-aside caching described in sections 2.1 and 2.2, respectively, of Bronson et al. [7].

5.1 Metastability due to GC

In this section, we develop a small-scale reproduction of the GC metastable failure seen in Section 4. This allows us to perform controlled experiments to validate the sustaining effect and study the factors that affect vulnerability. We confirm that GC can cause metastability and that the vulnerability increases with load. Since the sustaining effect is due to a high queue length causing memory pressure and GC slowdowns, we find that the memory size also impacts the degree of vulnerability.

5.1.1 Experiment Setup

Our reproduction is a multi-threaded java program compiled via JDK 8 under default GC settings except we experiment with MaxHeapSize. Each thread processes a job consisting of many memory allocations. Each job allocates a 0.5MB array of arrays and then proceeds to allocate each row in this 2D array, adding an additional 0.5MB of data. Once a job completes, the allocated memory is unreferenced and will eventually be garbage collected. The main thread launches jobs following a Poisson process with a configured request rate measured in requests per second (RPS). We launch the java program in a docker container configured with 1GB of memory running on an AWS EC2 m5.large instance.

5.1.2 Inducing Metastable Failures

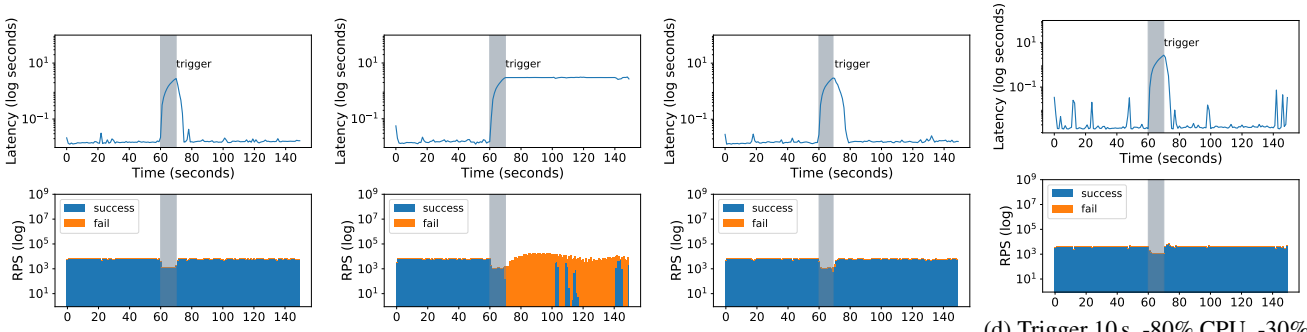
To illustrate the metastability, we vary RPS over time and plot the relevant metrics in Figure 4a. The initial RPS increase causes queue length and the GC duration to increase. Even as RPS is reduced over time, the sustaining effect causes the queue length and GC duration to remain high.

To gain a deeper understanding of the sustaining effect that causes the metastability, we extract detailed metrics from GC logs. Figure 4b shows that queue length, which we directly measure from arrival/completion timestamps, is correlated with GC duration. This is because there are more active objects to process during a GC cycle when there's a high queue length, and there is higher memory pressure as well. The figure shows a scatterplot of the normal behavior, though we see a similar correlation during metastable failures as well.

Figure 4b also shows that GC causes the application to pause, which slows down the jobs. Here, we configure the JVM to print a more detailed metric (PrintGCApplicationStoppedTime) to indicate how the JVM impacts the job's running time. We find that GC activity is causing the application to pause and slow down. As a result, the application isn't able to process jobs as efficiently, resulting in a higher queue length, thus completing the feedback cycle.

We next study the factors that affect vulnerability by exposing the example to varying trigger sizes. In our example, we generate triggers by injecting 100% stalls in the program for varying trigger durations. During the trigger, requests still arrive, but are not launched and do not begin processing. Once the trigger completes, there is effectively a large burst of backlogged requests that creates a large spike in the queue length until the backlog is handled. In our model, this corresponds to the bottom left scenario in Figure 1 where a load spike trigger (i.e., L_{org} increases) starts a capacity degradation amplification (i.e., C_{sys} degrades) due to GC.

Figure 4c shows how the vulnerability varies as a function of RPS. At high RPS, even small delays would cause the system to fall into a metastable failure state, whereas at low RPS, the system can mostly recover unless there is a very large trigger duration. The figure also shows how the vulnerability changes with the JVM memory size. Striped areas show regions where the metastability depends on the higher or lower memory size. For example, the striped region between the max heap (i.e., JVM memory) sizes indicates it is a metastable failure region for the smaller size and a vulner-



(a) Trigger 10 s, -78% CPU. (b) Trigger 10 s, -80% CPU. (c) Trigger 9 s, -80% CPU. (d) Trigger 10 s, -80% CPU, -30% RPS.

Figure 5: Metastability in a Replicated State Machine (RSM) due to retries.

able or stable region for the larger memory size (depending on RPS). Larger memory sizes decrease the memory pressure, which lowers the effect of GC. Thus, the system is less vulnerable with more memory and can sustain higher trigger durations and higher RPS. Nevertheless, the system is still subject to metastable failures, so understanding the degree of vulnerability is important for managing the system.

5.2 Metastability due to Retries

We next demonstrate an example of a metastable failure in a replicated state machine (RSM) model utilizing a popular NoSQL database. RSMs are prone to slowdowns [28,41] that can act as capacity-decreasing triggers for metastable failures. We use the slowdowns of varying magnitude and duration to induce metastable failures where retries create the sustaining effect.

5.2.1 Experiment Setup

For this experiment, we rely on MongoDB replicated database [64] based on the Raft [42] replication protocol. We operate the database in a strongly-consistent mode in a cluster of 3 replicas. A primary and two secondary MongoDB servers (version 4.4.9) are deployed on AWS EC2 m5a.large instances with 2 vCPU and 8 GiB of RAM each using Docker containers. A client application provides a constant baseline workload of insert operations against the replicated MongoDB database. We deployed the client on a bigger m5ad.2xlarge instance with 8 vCPU and 32 GiB of RAM.

We keep the RSM in a vulnerable state by running a constant client workload of approximately 6,200 successful RPS. A client uses a 3-second timeout for requests and will retry each operation up to 4 times after the timeout. To introduce the slowdowns, we temporarily restrict the CPU resources on the docker container running the primary node. In our model, this corresponds to the top right scenario in Figure 1 where a capacity-decreasing trigger (i.e., C_{org} decreases) causes a workload amplification (i.e., L_{sys} increases) due to retries.

5.2.2 Inducing Metastable Failures

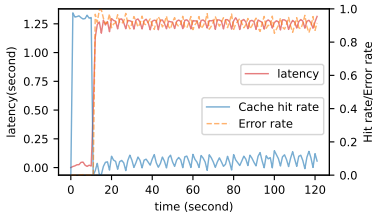
In Figure 5, we present the result of four experiments to demonstrate the relationship between trigger magnitude, trigger duration, and request rate. The figure truncates the experiments at 150 seconds, however, we ran the workloads for 500 seconds to ensure there is no delayed recovery from failure. We apply the capacity-decreasing trigger at the 60 second mark, as indicated by the gray shaded region in each of the subfigures.

(a) Baseline with no metastable failure. Figure 5a demonstrates a trigger of 10 seconds with a 78% reduction in CPU availability. This trigger briefly reduced the success rate of client requests, as observed by the dip in throughput with a corresponding increase in latency. The impact was brief with the occurrence of limited failures and retries towards the end of the trigger duration. The system does not enter a metastable failure state and recovers shortly after the trigger is removed.

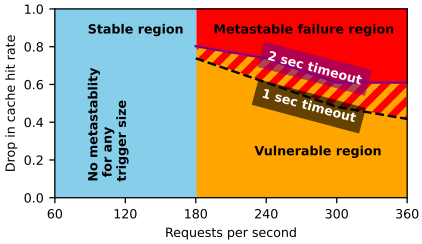
(b) Increased trigger magnitude causes metastable failure. Figure 5b demonstrates a metastable failure in an RSM. This result illustrates that even a slight increase in a trigger magnitude can push the system into metastability. In this instance, the trigger is of the same duration (10 seconds) against the same workload as (a) but with an 80% reduction in CPU availability (2% additional reduction). With this trigger magnitude, the system performance does not ever recover once the trigger is removed. Latency plateaus at approximately the client timeout of 3 seconds, and the total number of attempted requests peaks at around 20,000 RPS (a 3× increase over baseline), and goodput is reduced by ≈ 90% to 600 RPS. The client retry mechanism provides the feedback loop that prevents the system from resuming a normal state.

(c) Decreased trigger duration averts metastable failure. Figure 5c demonstrates that a minor change to a trigger duration, compared to the previous experiment (Figure 5b), can prevent a system from entering the metastable failure state. The experiment setup is the same as (b), except the trigger duration is reduced by 1 second from 10 seconds to 9 seconds. Similar to (a), we observe a transient increase in latency and a corresponding reduction in goodput. However, the system's performance recovers in this experiment, demonstrating the impact of trigger duration on vulnerability.

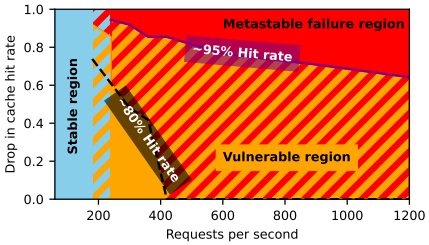
(d) Reduced load averts metastable failure. Figure 5d illustrates system performance when the base workload is reduced to about 4,200 RPS (≈ 30% lower than the baseline RPS) to show a system with more idle resources to handle triggers. We used the same trigger magnitude and duration as



(a) Trigger at 10s induces a metastable failure where a low cache hit rate causes the database to become overloaded. This results in high sustained error rates and latencies.



(b) Vulnerabilities for different request timeouts. Striped area between lines indicates it is a metastable failure region for 1 second timeouts and a vulnerable region for 2 second timeouts.



(c) Vulnerabilities for different steady state cache hit rates. Striped areas indicate the region is different for the hit rates (e.g., metastable failure for ~80% and vulnerable for ~95%).

Figure 6: Metastability in Look-aside cache.

in (b) (a trigger intensity of 80% and a trigger duration of 10 seconds) that pushed a more loaded system into a metastable failure. With more idle resources to handle the transient performance degradation, the system handled the trigger gracefully with only a temporary increase in latency.

Throughout our experiments on a replicated database, we have established that a trigger that sets off the retry process can lead to the feedback loop that prevents a distributed system from recovering. Moreover, even small changes to a trigger had a significant impact—a 2%-decrease in available CPU or a 1-second increase in duration separated successful recovery from a metastable failure. All experiments exhibit a small number of failed requests immediately before the trigger is removed. However, (b) demonstrates an increased level of failures and retries during the last second of the trigger. This suggests that timely removal of the trigger can prevent the transition into the metastable failure state.

5.3 Metastability due to Look-aside Cache

We next illustrate another type of metastable failures due to look-aside caching. Look-aside caching is a popular caching strategy where an application looks for data in a cache and will retrieve data from a backend system for cache misses. The application is then responsible for putting the data from the backend into the cache.

The metastability arises because the application is not always able to add the data from the backend into the cache. Specifically, if a trigger causes the cache hit rate to drop, then that would result in a higher rate of misses and an unexpectedly high rate of requests to the backend system. This amplified workload would in turn cause the backend to slow down, which would lead to timeouts in the application and/or backend. When there are timeouts, the application is unable to put data into the cache. As a result, the cache hit rate remains low, which sustains the metastability.

5.3.1 Experiment Setup

To replicate this metastability, we build an example web application with a MySQL database (34.6 million entries, totaling 15GB) and a memcached cache (1GB). If the web application is unable to find an item in the cache, it queries the database and stores the result in the cache. Items are requested following a Zipf distribution—a common distribution for representing cache entry popularity [6].

The arrival times are generated via a standard Poisson process at the desired RPS from our load generator. Our web server runs a standard Nginx + PHP setup, and we configure 1 second timeouts for the requests, which are much higher than the normal request processing times.

5.3.2 Inducing Metastable Failures

Figure 6a shows an example of a metastable failure when a trigger causes the hit rate to unexpectedly drop at time 10s. We see that the backend traffic sharply increases, which results in timeouts and errors. Since the application is unable to get the data before the timeout, no new data is added to the cache, which sustains the low hit rate for long periods of time.

We next run the system under different RPS and inject triggers of different magnitudes to evaluate whether the system is able to recover. We inject triggers by deleting the hottest items in the cache*. In our model, this corresponds to the bottom right scenario in Figure 1 where a capacity-decreasing trigger (i.e., C_{org} decreases) starts a capacity degradation amplification that causes the degradation to persist even after the cache memory is available for use (i.e., C_{sys} remains degraded). After the trigger, we run the system for an hour to see if it can recover or if the metastable failure persists. If the system doesn't recover within an hour, we mark this as a metastable failure. Caching systems by nature are self-healing, and we would expect the system to eventually recover if there's a non-zero chance that a request would successfully add data to the cache. However, long-term outages are catastrophic to companies so we still deem these cases as metastable failures.

Figure 6b illustrates the different degrees of vulnerability in our look-aside caching example. Under low RPS, the system is stable and can recover even if the entire cache is wiped. As the RPS increases, the system becomes more vulnerable where smaller drops in hit rate could cause the system to fall into a metastable failure region and not recover.

Figure 6b also illustrates the impact of the request timeout parameter. When increasing the timeout from 1 second to 2 seconds, the vulnerability at each RPS is decreased (i.e., a higher trigger magnitude is needed to cause metastable fail-

*Dropping the hottest items gives a conservative bound on the metastable region since these are the easiest items to recover.

ures). So there is a trade-off with setting the request timeout—a higher timeout decreases the vulnerability, but it takes longer to detect failed requests, whereas a lower timeout can quickly detect issues, but increases the metastable vulnerability.

Figure 6c demonstrates the impact of the steady state cache hit rate on vulnerability. When comparing a workload with a $\approx 80\%$ cache hit rate vs. a workload with a $\approx 95\%$ cache hit rate, we see that the higher hit rate is less vulnerable. This is because the workload has a more skewed popularity distribution where a small number of keys constitutes a large fraction of the requests. This skewness makes it easier to recover from a drop in hit rate. However, higher hit rates enable the system to operate at higher RPS where the system is vulnerable, and we see that the ~95% workload has a much wider range of vulnerability in terms of RPS. Thus one still needs to consider metastable issues at high hit rates.

6 Discussion

6.1 Multi-System Failures

Many metastable failures involve a combination of systems or components interacting together. Often, these failures are described as cascading failures (GGL1), where the failure of one system causes further faults in other components. The interactions between systems make it more difficult to identify the sustaining effect and enact quick fixes. Our caching example is a good illustration of such a multi-system failure.

In our caching example, the cache and storage systems are coupled together. When a cache fails, the result is a load spike in the storage system – a capacity degradation of one component cascades to a load increase in another. Even in the absence of workload amplification, this multi-system example has a sustaining effect. The complete cache-storage system needs the storage component to respond in time to fill the cache and reduce the load on storage. At the same time, the storage cannot do so due to the overload, creating a sustained condition where the overload cannot be alleviated even after the cache has all servers back up again.

6.2 Human Factors

Around 50% of the observed triggers have some direct human involvement, such as the deployment of buggy configuration (GGL3, GGL4, AWS1), rushed testing and deployment (AZR4), incomplete testing that fails to find bugs (GGL2, GGL4, AZR2, AZR3, IBM1), and regular maintenance (ELC1). For instance, in the AZR4 incident, engineers rushed a buggy code for deployment without proper testing. The bug would increase CPU consumption on some background tasks, essentially decreasing the system's processing capacity. Moreover, the deployment was happening on Friday before a long holiday weekend when the load on the system was lower than usual, potentially preventing the deployment procedure from catching the capacity degradation. After the holiday weekend when traffic returned to normal, the system was overwhelmed, which increased latency, caused timeouts, and failed user requests. This issue could have been avoided with more complete testing and better deployment practices. Another example of a human factor in metastability is the GGL4 incident where engineers bypassed the testing phase and released a buggy configuration to production.

6.3 Fix to Break

Misunderstanding the processes that cause the failure can lead engineers to adapt long-term fixes or changes that can further exacerbate the vulnerability for metastable failures. For example, not realizing the existence of a feedback loop may cause engineers to introduce changes that make the feedback loop more severe. In the AWS2 incident that brought down AWS SimpleDB, the storage servers frequently communicate with the locking service to ensure they are still part of the system. When an overload to the locking service occurred, the storage servers started to timeout and retry, further adding to the locking service overload. After several retries, the servers would demote themselves and stop serving the storage workload. The locking service remained overloaded for as long as enough storage servers were alive to keep the lock service busy. In the aftermath of the incident, engineers decided that servers must continue to retry the locking service instead of giving up, as the lack of prolonged retries was seen as the reason for botched recovery. Unlimited retries, however, can put a lot more workload amplification on the system and make the sustaining effect more severe. A similar incident (AWS3) happened to the DynamoDB database about a year later. The storage nodes did not back out of retrying to get updated membership data, causing a massive workload amplification and metastable failure.

Another example of this is the SPF1 and SPF2 incidents. In the aftermath of the first incident, engineers added significant logging to the error path of request execution to better understand the cause of the load spikes and retries. In SPF2, the additional logging after a load spike and initial retries increased the cost of each retry, adding more load to the system and causing more requests to retry.

6.4 Mild Metastable Failures

Many metastable failures are severe enough to cause a significant service disruption. However, this is not necessarily the case for all metastable failures. The CAS1 incident is an example of metastable behavior that did not cause a significant outage. Another example is our Twitter case study. While the metastable failure was severe enough to trigger internal alerts, it was very far from becoming an outage. This mildness was partly due to monitoring of key performance metrics and a timely response.

6.5 Prevention and Mitigation

A crucial aspect of preparing for metastable issues is understanding the system's vulnerability. As we have seen throughout our experiments with retries, caching, and GC, many factors impact the vulnerability of a system, ranging from the load to trigger magnitude and duration and to sustaining effect mechanisms, such as workload amplification growth. With a proper understanding of the processes involved, we can

have better control over both the triggers and sustaining effects. For example, our Twitter case study showed that even if the sustaining effect cannot be eliminated, knowing and understanding its characteristics can help engineers adjust parameters and reduce its impact in the future.

Similarly, systems may not be able to avoid all possible triggers, but they can often mitigate the trigger's impact. For instance, designing systems to be more resilient to component slowdowns [41] can help reduce the severity of triggers and reduce the system's vulnerability. Designing automated mitigation strategies can reduce the trigger duration, which result in a small performance blip instead of a metastable failure.

While autoscaling of resources can help mitigate metastable failures in some cases, it does not necessarily prevent metastable failures. Autoscaling is a way to increase the normal capacity C_{norm} of a system in response to load events, which should also raise the stable threshold C_{stable} and help the system recover sooner. However, autoscaling can be extremely costly for large systems and large work amplification factors. For instance, a loss of a cache with 99% hit-rate can result in a 100X amplification. Whether the current autoscaling techniques can scale up fast enough to avert a metastable failure requires further research. Furthermore, it is not always possible to autoscale services due to stateful components and system complexity (e.g., case Azure LL1H-9CZ).

7 Related Work

Since metastable failures were established as a class very recently [7], there have not yet been any studies particularly about them. However, researchers have discovered other classes of failures that we think are relevant to metastability. Specifically, the types of failures and bugs that we discuss below often act as triggers that lead to metastable failures.

One such class is *fail-slow* failures [27], which were extensively studied under different names: *fail-stutter* [3], *gray-failure* [29], and *limpware* [15, 16, 25]. Fail-slow failures happen when a hardware experiences a significant slowdown but is still functional. Since fail-slow failures can occasionally exhibit transient stops [27], they can trigger metastable failures. Unlike metastable failures, however, fail-slow failures are essentially subtle hardware failures that can be fixed by replacing the faulty hardware.

Another related class of failures is due to scalability bugs [34, 53]. These are latent software bugs that are scale-dependent—they only surface in large-scale deployments and are not discoverable in small-scale testing. As a result, load spikes can expose scalability bugs, which can trigger metastable failures. We have observed several incidents where load spikes exposed a bug that triggered a metastable failure.

Finally, there have been multiple studies on failures in distributed systems caused by configuration changes [40, 59] and software upgrades [63] both of which were predominant triggers of metastable failures in our study.

In general, most prior studies classify incidents according to their main root cause, for example, software bugs, hardware faults, misconfiguration, etc. The metastable failure model, where a service in the vulnerable state is tipped over to failure by a trigger, allows a richer, multi-dimensional characterization of bugs. Metastability would likely explain some of the bugs others have studied, but to date, researchers have lacked a framework for identifying such failures. It is notable that in [61] the authors observe that failures often "require an unusual sequence of multiple events with specific input parameters from a large space", which suggests that they may have in fact encountered metastable failures.

The cloud outage study of [26], which examines almost 600 publicly reported outages in popular Internet services, discusses the idea of "hidden single points of failure" and observes that the recovery process itself is often faulty or simply doesn't run because the right metrics are not being monitored. Our model for metastable failures may help identify the metrics that may act as triggers. They also note that the recovery process can be a source of metastable amplification, such as with retry storms or failover to cold caches.

In the Azure incidents studied by Liu et al. [36], running-environment mitigation techniques are commonly applied, such as restarting or migrating processes or adding capacity resources. The authors note that to date there has been little work on automation of such recovery methods – this would also be a fruitful direction in mitigating metastable failures.

What sets a metastable failure apart from all of the above is that its root cause is not a specific hardware failure or a software bug. It is an emergent behavior of a complex system that naturally arises from optimizations for the common case. Specifically, if the aforementioned failures do not trigger a metastable failure, then identifying and eliminating them restores the system functionality. If, however, they do trigger a metastable failure, then eliminating them will not restore the system's functionality.

8 Conclusion

Metastable failures are a class of system failures characterized by sustaining effects that keep systems in a degraded state and resist recovery. While relatively infrequent, metastable failures were behind big outages at large internet companies (including a recent AWS outage on December 7th, 2021). In this work, we confirm this observation by studying public incident reports. We then extend the metastability framework based on our observations for a more accurate metastability model. We validate our model by building three applications and reproducing different instances of metastability on them. We hope our work spurs further research into understanding and preventing metastable failures.

Acknowledgments

We thank our shepherd Atul Adya and the anonymous reviewers who provided constructive and helpful feedback. We also thank Nathan Bronson for his insightful comments and suggestions. This research was supported in part by AWS Cloud Credit for Research.

Appendix

A Proof of model theorems

A.1 Proof of Theorem 1

Assuming no overloading trigger, then by Definition 2 we have $C_{norm} - L_{norm} > m_{trigL} + m_{trigC}$. So,

$$\begin{aligned} L_{org}(t) &\leq L_{norm} + m_{trigL} && \text{(Definition 1)} \\ &< C_{norm} - m_{trigC} && \text{(assumption)} \\ &\leq C_{org}(t) && \text{(Definition 1)} \end{aligned}$$

Since $L_{org}(t) < C_{org}(t)$ for all t, then $\Delta t_{trig} = 0$. Since $w_L(0) = 1$ and $w_C(0) = 1$, then $\alpha_L(t) = 1$ and $\alpha_C(t) = 1$ for all t. Therefore, $L_{sys}(t) = L_{org}(t) < C_{org}(t) = C_{sys}(t)$ for all t by Definition 4. Thus, the system is never overloaded and never in a metastable failure state. □

A.2 Proof of Theorem 2

Assume $L_{norm} < C_{stable} = \frac{C_{norm}}{(w_L^* * w_C^*)}$. So,

$$\begin{aligned} L_{norm} * \alpha_L(t) &\leq L_{norm} * w_L^* && \text{(Definition 4)} \\ &< C_{stable} * w_L^* && \text{(assumption)} \\ &= C_{norm} * w_L^* / (w_L^* * w_C^*) && \text{(assumption)} \\ &= C_{norm} / w_C^* && \text{(algebra)} \\ &\leq C_{norm} * \alpha_C(t) && \text{(Definition 4)} \end{aligned}$$

Thus, under the normal conditions without triggers, the amplification factors are bounded such that the system is always stable even with the worst-case amplification factors. □

A.3 Proof of Theorem 3

Assume $w_L(\Delta t_{trig}) * w_C(\Delta t_{trig}) < \frac{C_{norm}}{L_{norm}}$. So,

$$\begin{aligned} L_{norm} * \alpha_L(t) &\leq L_{norm} * w_L(\Delta t_{trig}) && \text{(Definition 4)} \\ &< C_{norm} / w_C(\Delta t_{trig}) && \text{(assumption)} \\ &\leq C_{norm} * \alpha_C(t) && \text{(Definition 4)} \end{aligned}$$

Thus, under the normal conditions without triggers, the amplification factors are bounded such that the system is always stable. □

A.4 Proof of Theorem 4

Assume at time t, $L_{sys}(t) - C_{sys}(t) \geq \alpha_L(t) * m_{trigL} + \alpha_C(t) * m_{trigC}$. So,

$$\begin{aligned} & L_{norm} * \alpha_L(t) \\ &= L_{org}(t) * \alpha_L(t) - (L_{org}(t) - L_{norm}) * \alpha_L(t) && \text{(algebra)} \\ &\geq L_{org}(t) * \alpha_L(t) - m_{trigL} * \alpha_L(t) && \text{(Definition 1)} \\ &= L_{sys}(t) - m_{trigL} * \alpha_L(t) && \text{(Definition 4)} \\ &\geq C_{sys}(t) + \alpha_C(t) * m_{trigC} && \text{(assumption)} \\ &= C_{org}(t) * \alpha_C(t) + \alpha_C(t) * m_{trigC} && \text{(Definition 4)} \\ &\geq C_{org}(t) * \alpha_C(t) + \alpha_C(t) * (C_{norm} - C_{org}(t)) && \text{(Definition 1)} \\ &= C_{norm} * \alpha_C(t) && \text{(algebra)} \end{aligned}$$

Thus, if at time t we removed the triggers and reverted to the normal load and capacity, then the amplifying factors would cause the system to remain in an overloaded state. So the system is in a metastable failure state. □

References

[1] Anonymous. Overload because of hint pressure + MVs. Apache Cassandra Issue Tracker: https://issues.apache.org/jira/projects/CASSANDRA/issues/CASSANDRA-13810?filter=allopenissues, 2017.

[2] Azure Architecture Performance Antipatterns. Retry Storm antipattern. https://docs.microsoft.com/en-us/azure/architecture/antipatterns/retry-storm/, 2021.

[3] R.H. Arpaci-Dusseau and A.C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 33–38, 2001.

[4] Microsoft Azure. Azure status history. https://status.azure.com/en-us/status/history/, 2021.

[5] Betsy Beyer, Jennifer Petoff, Niall Richard Murphy, and Chris Jones. Site Reliability Engineering: How Google Runs Production Systems. https://sre.google/sre-book/table-of-contents/, 2016.

[6] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 1, pages 126–134 vol.1, 1999.

[7] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable Failures in Distributed Systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 221–227, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, LA, February 1999. USENIX Association.

[9] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the Straggler Problem with Bounded Staleness. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, page 22, USA, 2013. USENIX Association.

[10] CircelCI. DB Performance Issue Incident Report for CircleCI. https://circleci.statuspage.io/incidents/hr0mm9xmm3x6, 2015.

[11] IBM Cloud. Incident reports. https://cloud.ibm.com/status/incident-reports, 2021.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[13] Down Detector. Downtime Detector). https://downdetector.com, 2020.

[14] Availability digest article. Availability Digest). https://www.availabilitydigest.com/articles.htm, 2020.

[15] Thanh Do and Haryadi S. Gunawi. The Case for Limping-Hardware Tolerant Clouds. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)*, San Jose, CA, June 2013. USENIX Association.

[16] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[17] Data Dynamics. Data center dynamics). https://www.datacenterdynamics.com/en/news/?term=outages, 2020.

[18] Facebook. Solving the Mystery of Link Imbalance: A Metastable Failure State at Scale). https://engineering.fb.com/2014/11/14/production-engineering/solving-the-mystery-of-link-imbalance-a-metastable-failure-state-at-scale/, 2020.

[19] David Pobladori Garcia. Incident Management at Spotify. https://engineering.atspotify.com/2013/06/04/incident-management-at-spotify/, 2013.

[20] Jeremy M. Goldberg. The future of critical infrastructure is in the cloud. https://cloudblogs.microsoft.com/industry-blog/government/2021/10/25/the-future-of-critical-infrastructure-is-in-the-cloud/, 2021.

[21] Google. Google API infrastructure outage incident report. Google Developers blog: https://developers.googleblog.com/2013/05/google-api-infrastructure-outage_3.html, 2013.

[22] Google. Google App Engine Incident #19007. https://status.cloud.google.com/incident/appengine/19007, 2019.

[23] Google. Google Compute Engine Incident #19008. https://status.cloud.google.com/incident/compute/19008, 2019.

[24] Google. Google Cloud Infrastructure Components Incident #20005. https://status.cloud.google.com/incident/zall/20005, 2020.

[25] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[26] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 1–16, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 1–14, Oakland, CA, February 2018. USENIX Association.

[28] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Storage*, 14(3), oct 2018.

[29] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 150–155, New York, NY, USA, 2017. Association for Computing Machinery.

[30] The Wall Street Journal. Amazon Outage Disrupts Lives, Surprising People About Their Cloud Dependency. https://www.wsj.com/articles/amazon-outage-disrupts-lives-surprising-people-about-their-cloud-dependency-11638972001, 2021.

[31] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[32] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, July 1982.

[33] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. *SIGARCH Comput. Archit. News*, 44(2):517–530, mar 2016.

[34] Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi. Scalability Bugs: When 100-Node Testing is Not Enough. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 24–29, New York, NY, USA, 2017. Association for Computing Machinery.

[35] John D. C. Little. A proof for the queuing formula: L = λw. *Oper. Res.*, 9(3):383–387, jun 1961.

[36] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 155–162, 2019.

[37] Sean Lynch. Avoiding Death Spirals in Distributed Systems. https://blog.couchbase.com/avoiding-death-spirals-distributed-systems/, 2021.

[38] Aaron McDade. Significant Outage for Amazon Web Services Stalls Netflix, Delta Airlines, Others. https://www.newsweek.com/significant-outage-amazon-web-services-stalls-netflix-delta-airlines-others-1657077, 2021.

[39] Panagiotis Moustafellos and Ben Osborne. Elastic Cloud Incident Report: February 4, 2019. https://www.elastic.co/blog/elastic-cloud-incident-report-feburary-4-2019, 2019.

[40] Kiran Nagaraja, Fabio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[41] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 583–598. USENIX Association, November 2020.

[42] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[43] Byers Paddy. Cassandra counter columns: nice in theory, hazardous in practice. https://ably.com/blog/cassandra-counter-columns-nice-in-theory-hazardous-in-practice, 2021.

[44] peakscale + postmortem. Postmortem reports). https://pinboard.in/u:peakscale/t:postmortem/, 2020.

[45] Postmortems. Postmortems info). https://postmortems.info, 2020.

[46] Richard D. Schlichting and Fred B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, aug 1983.

[47] Amazon Web Services. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. https://aws.amazon.com/message/65648/, 2011.

[48] Amazon Web Services. Summary of the Amazon SimpleDB Service Disruption. https://aws.amazon.com/message/65649/, 2014.

[49] Amazon Web Services. Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region. https://aws.amazon.com/message/5467D2/, 2015.

[50] Amazon Web Services. AWS Post-Event Summaries. https://aws.amazon.com/premiumsupport/technology/pes/, 2021.

[51] Amazon Web Services. Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region. https://aws.amazon.com/message/12721/, 2021.

[52] Isabella Steger. How Amazon Outage Left Smart Homes Not So Smart After All. https://www.bloomberg.com/news/articles/2021-12-08/amazon-outage-sparks-anger-as-fridges-stop-people-locked-out, 2021.

[53] Cesar A. Stuardo, Tanakorn Leesatapornwongsa, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, Wei-Chiu Chuang, Shan Lu, and Haryadi S. Gunawi. ScaleCheck: A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 359–373, Boston, MA, February 2019. USENIX Association.

[54] Thousandeyes. Internet Outages Map. https://www.thousandeyes.com/outages/, 2020.

[55] SRE Weekly. SRE Weekly Digest. https://sreweekly.com/about-sre-weekly-2/, 2020.

[56] AWS Well-Architected. Design Interactions in a Distributed System to Mitigate or Withstand Failures. https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/design-interactions-in-a-distributed-system-to-mitigate-or-withstand-failures.html, 2021.

[57] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 230–243, New York, NY, USA, 2001. Association for Computing Machinery.

[58] Wikitech. Incident documentation/2021-03-30 Jobqueue overload. https://wikitech.wikimedia.org/wiki/Incident_documentation/2021-03-30_Jobqueue_overload, 2021.

[59] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 244–259, New York, NY, USA, 2013. Association for Computing Machinery.

[60] David Yanacek. Using load shedding to avoid overload. https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/, 2021.

[61] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, Broomfield, CO, October 2014. USENIX Association.

[62] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 29–42, USA, 2008. USENIX Association.

[63] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. *Understanding and Detecting Software Upgrade Failures in Distributed Systems*, page 116–131. Association for Computing Machinery, New York, NY, USA, 2021.

[64] Siyuan Zhou and Shuai Mu. Fault-Tolerant replication with Pull-Based consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 687–703. USENIX Association, April 2021.

Demystifying and Checking Silent Semantic Violations in Large Distributed Systems

Chang Lou Yuzhuo Jing Peng Huang

Johns Hopkins University

Abstract

Distributed systems today offer rich features with numerous semantics that users depend on. Bugs can cause a system to silently violate its semantics without apparent anomalies. Such silent violations cause prolonged damage and are difficult to address. Yet, this problem is under-investigated.

In this paper, we first study 109 real-world silent semantic failures from nine widely-used distributed systems to shed some light on this difficult problem. Our study reveals more than a dozen informative findings. For example, it shows that surprisingly the majority of the studied failures were violating semantics that existed since the system's first stable release.

Guided by insights from our study, we design Oathkeeper, a tool that *automatically* infers semantic rules from past failures and enforces the rules at runtime to detect new failures. Evaluation shows that the inferred rules detect newer violations, and Oathkeeper only incurs 1.27% overhead.

1 Introduction

Users' increasing reliance on distributed systems highlights the importance of ensuring they work correctly. Unfortunately, real-world distributed systems inevitably encounter failures. When a failure is recognizable through explicit signals such as crash, timeout, error code, or exception, timely actions can still be taken to detect [22,40,46] and mitigate [41,52,53] the failure. A vexing problem occurs when a system is operational but *silently* breaks its semantics without apparent anomalies.

Take a distributed notification service as an example, which provides an interface that promises to invoke the client callback whenever the status of some object changes. A bug may cause this system to miss invoking the callback upon a change or invoke the callback more than necessary. As another example, a distributed file system that is supposed to replicate data blocks by user-configured n copies may incorrectly under-replicate some blocks without any explicit errors.

Such failures can lead to severe consequences because they violate the guarantees a system provides to its users. They also break the contracts that other components or applications rely on, and result in amplified incorrectness. Moreover, since the violation is silent, the damage exacerbates over time. For example, as the buggy distributed file system that silently violates its replication policy continues to run, more and more newly created files will be subject to potential data loss.

System	Ver.	Client API	Public Method	Admin Command	Config.
ZooKeeper	3.4.6	38	219	13	30
ZooKeeper	3.6.2	78	2,853	18	128
HDFS	2.7.2	128	5,293	11	224
HDFS	2.10.0	162	6,306	12	449
Kafka	2.6.0	166	2,661	76	366
Kafka	2.8.0	171	3,107	86	379

Table 1: Number of public interfaces in popular distributed systems. An interface can have multiple semantics under different settings.

Distributed systems today have rich semantics (Table 1) exposed through client APIs, public methods including RPCs among internal components, administrator commands, configuration parameters, *etc.* One interface often encodes multiple guarantees. New interfaces and semantics are also continuously introduced as a system evolves. These characteristics together make it challenging to ensure that a distributed system conforms to its semantics in production settings.

Indeed, real-world evidence shows that semantic violations occur in practice. In a Google cloud incident [3], a traffic engineering subsystem that is supposed to throttle traffic upon congestion incorrectly throttled traffic even though the network was not congested. Another highly-impactful global outage [2] was caused by a quota system incorrectly reporting the usage for a user ID service as zero.

However, other than anecdotal evidence, the problem of silent semantic violations in distributed systems remains mysterious, despite its severe consequences. For instance, mature distributed systems include extensive test cases to check the correctness of their features. Thus, it is natural to assume silent semantic violations are rare in production because testing likely has eliminated most of them. In addition, while adding assertions and runtime verification [43,44,48,57] are potential solutions, the conventional wisdom is that they are expensive and semantic rules are difficult to get. It is also unclear what kind of semantics are violated in practice.

To systematically understand this problem, we present, to our best knowledge, the first empirical study on 109 *real-world* silent semantic violations from nine widely-used distributed systems. Through these cases, we analyze key questions such as *how prevalent are semantic violations in practice*, *what semantics are violated*, *why are these failures not caught in testing*, and *how are these silent violations detected*.

Our study provides quantitative data points to answer these

questions. The study findings also challenge some conventional wisdom and reveal gaps in the current practice. We highlight several findings:

- Contrary to the belief that silent semantic violations rarely occur in deployed systems, they have significant presence (39%) among sampled failures of all kinds.
- While the studied systems get more extensively tested over time and continue to add new features and semantics, their initial semantics do *not* become more bulletproof. On the contrary, more than two thirds of the failures violate semantics that have existed since the system's first stable release.
- Although these are distributed system failures, most (74%) violations can be determined locally in some component.
- The violated semantics are often *not* untested but rather well covered by existing test cases.
- Enabling assertions in release builds helps by converting semantic violations into crash failures. One studied system does this and has the lowest ratio of semantic failures.
- In many cases, although a semantic was initially honored, it was later violated, thus one-time assertions are insufficient.
- Many system semantics are vulnerable to violations during maintenance operations or node events.

Given the prevalence (as our study indicates) and severity of silent semantic violations, we design a tool Oathkeeper to help users check silent semantic violations at runtime. The tool design is directly guided by insights from our study.

Specifically, we find that in 73% of the cases, developers add regression tests after the failure is reported, which contain valuable information about the failed semantic. However, the majority of the studied cases still violate semantics that have been tested before. A major reason for the gap is that these regression tests are usually patch-driven: they only check if the specific bug is fixed in a particular setup using a bug-triggering workload. The underlying semantics can continue to be broken with different root causes in different scenarios.

Based on this insight, Oathkeeper leverages the regression tests and tries to infer the underlying semantic rules implied by the tests. To do so, Oathkeeper runs the tests on both the buggy version and patched version of the system, and takes a *template-driven* approach to automatically infer semantic rules from the two traces. Oathkeeper then deploys these semantic rules to production to catch future violations that are caused by different bugs under different conditions.

We evaluate Oathkeeper on ZooKeeper, HDFS, and Kafka. Oathkeeper infers hundreds to thousands of semantic rules from the old regression tests in these systems. With the inferred rules, we evaluate Oathkeeper on seven real-world semantic failures that were introduced long (9–34 months) after the old failures. Oathkeeper detects violations for six of them. With all rules enabled, Oathkeeper on average only incurs 1.27% throughput overhead to the target systems.

The contributions of this paper are two-fold: (i) the first study on *real-world* silent semantic violations in nine popular distributed systems; (ii) the design of Oathkeeper, which automatically infers semantic rules for large distributed systems to check silent semantic violations at runtime.

The source code of Oathkeeper is publicly available at:

`https://github.com/OrderLab/OathKeeper`

2 Background

2.1 Definition

We consider a distributed system $\mathcal{S}$ that provides services through a collection of operations. Each operation o has certain *semantics* [29]. The semantics encode guarantees that o makes about the output, system states, and results of subsequent operations, in response to some triggering condition c. The condition c can be a client request, an admin command (at the server side), a message from internal components, as well as an environment change including the passage of time. The semantics of $\mathcal{S}$ are all the guarantees provided by the history of operations $\mathcal{S}$ executes in response to a list of c.

A semantic violation (failure) occurs when $\mathcal{S}$ breaks some of its semantics in an execution. The failures may exhibit explicit error signals, such as crashes, timeouts, and exceptions. In such cases, the violations overlap largely with existing failure models and can be well addressed by existing techniques.

This work focuses on *silent* semantic violations, in which $\mathcal{S}$ violates its semantics but remains operational without exhibiting explicit error signals ($\mathcal{S}$ is unaware of its misbehavior). We focus on this class of failures because they are under-studied yet incur damaging consequences, and they pose significant challenges to testing, failure detection, and recovery.

Silent semantic violations differ from other failure modes in observability. Fail-stop failures cause complete loss of functionality, which can be observed with simple measures such as monitoring heartbeats. Fail-slow [32], partial failures [46] and gray failures [37] only cause some functionality to be broken (slow). But these issues can still be observed with generic approaches, *e.g.*, checking exceptions or timeouts [45]. In comparison, silent semantic violations are difficult to observe without a deep understanding of $\mathcal{S}$' semantics and execution.

Another way to interpret the "silent" aspect is on the semantics being violated. If $\mathcal{S}$ only has a few operations, all of which have well-defined and thoroughly checked semantics, semantic violations in $\mathcal{S}$ will be observable failures. Unfortunately, distributed systems have a large number of interfaces (Table 1), many of which have loosely-defined (or hidden) semantics that cannot be easily checked. Consequently, violations of such semantics are difficult to detect and address.

2.2 An Example

We show an example of silent semantic failures from our study (Section 3). ZooKeeper is a coordination service with a hierarchical data model. Its clients store data by creating `znode` in a namespace. A special type of `znode` is called ephemeral node. The semantics of the ephemeral node `create()` operation guarantees that the `znode` exists for as long as the creating

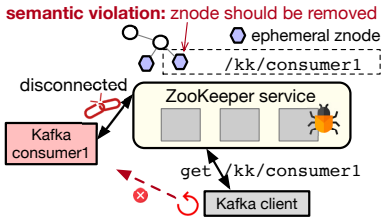


Figure 1: A silent semantic failure in ZooKeeper.

client's session and will be deleted once the associated session ends. The triggering conditions are the create request and the client session disconnection. Ephemeral nodes are commonly used to store membership information. For example, HDFS implements its leader election using ephemeral nodes [27].

In a production ZooKeeper failure [13], some ephemeral node still existed even though the client session that created them was long gone. Specifically, a Kafka consumer crashed but the associated `znode` was not deleted (Figure 1). As a result, when Kafka clients queried ZooKeeper to discover consumer information, they kept trying to connect to the crashed consumer. In other settings, this semantic violation can propagate to other dependent applications, *e.g.*, it will break HDFS namenode's automatic fail-over feature, which depends on the ephemeral node semantics, causing an HDFS service outage.

3 Study Methodology

Compared to other failure modes in distributed systems, silent semantic violations are not well understood. To fill this gap, we conduct a study on *user-reported* silent semantic failures from nine large-scale distributed systems (Table 2). We select these systems because they are representative, mature, widely used in production, and record many user-reported failures.

To collect the failure cases, we first query the study systems' issue trackers to find tickets that (1) are marked as "bugs", (2) have priorities higher than "minor", (3) are resolved, (4) involve the server components. This step returns a large number of tickets. We then *randomly* sample a subset (Table 2). Among this subset, some are not real failures, such as issues found in internal testing. The remaining ones (*valid* column in Table 2) are potential production failures. We then read their descriptions and check whether the failures violate system semantics. We filter crashes, aborts, out-of-memory errors, and semantic failures with clear error signals.

After the above step, we get a candidate set of production silent semantic failures (*Candidate* column). Due to time constraints, we perform in-depth analyses on a subset of the candidate cases, preferring those with sufficient information and discussions. This gives us the final study dataset (*Studied* column) of 109 production semantic failure cases.

Note that our sample sizes vary across systems. This is because the studied systems' tickets vary greatly in terms of their information, quality, and bug types. If using a fixed sample size or ratio, one system can dominate the study and produce extremely biased findings. Our sampling instead is done iteratively: for a particular system, if after an initial sampling, its number of *Candidate* cases is too small or 0, we sample more, until the candidate numbers for different systems are relatively balanced. Note that each iteration in this process is still randomly choosing from the *All* tickets.

System	Category	Lang.	All	Sampled (valid)	Candi -date	Stud -ied
Cassandra (CS)	Database	Java	3,308	69 (54)	25	12
CephFS (CF)	File Sys.	C++	673	673 (123)	37	12
ElasticSearch (ES)	Search	Java	4,101	101 (46)	26	10
HBase (HB)	Database	Java	6,143	233 (80)	32	14
HDFS (HF)	File Sys.	Java	3,409	99 (52)	22	14
Kafka (KF)	Streaming	Scala	2,764	142 (92)	39	13
Mesos (ME)	Cluster Mgr.	C++	2,462	116 (47)	21	12
MongoDB (MG)	Database	C++	14,776	355 (151)	30	10
ZooKeeper (ZK)	Coordination	Java	1,141	134 (102)	36	12
Total			38,786	1,922 (747)	268	109

Table 2: Studied systems, the tickets (of various kinds) in the issue tracker of each system, the cases we sampled, and cases studied.

Threats to Validity. Like all empirical studies, our study is subject to validity problems such as the representativeness and biases. We cover popular distributed systems of different types, such as database, file system, and search engine, to improve the representativeness. To minimize selection bias, we *randomly* sample the cases. We also spread the sampling across times so we are not biased by some specific version. To reduce the manual inspection errors, we write a detailed analysis document for each case and have multiple inspectors examine each document to reach a consensus.

Although our study provides informative findings on semantic failures in the studied systems, they may not be generalized to other systems beyond the scope this study was conducted. Our study is also biased by programming languages (Java and C++); the findings may not generalize to systems written in other languages such as Erlang or Elixir, which embrace "let-it-crash" error handling philosophy [18].

4 Are Silent Semantic Failures Rare?

Prevalence. An important question about silent semantic violations is whether they occur rarely in production. Getting accurate prevalence data requires examining thousands of tickets for each system, which is a daunting task. We instead obtain an approximate result by calculating the percentage of silent semantic failures in our sample set. Specifically, we calculate the percentages of the number of *candidate* cases in Table 2 over the number of *valid* cases in the sample. Note that the candidate cases are examined to be indeed silent semantic failures, even though we only study a subset of them.

Finding 1: *Silent semantic failures have significant presence across all studied systems, occupying 20%–57% (39% on average) of the sampled cases for all types of failures.*

The percentages vary in different systems. Systems such as ElasticSearch and Cassandra have a higher percentage of semantic failures (57% and 46%, respectively). MongoDB has the lowest ratio (20%). We will discuss in Section 8 these

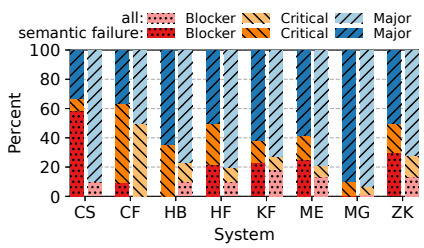


Figure 2: Issue priorities of semantic failure cases and all valid sampled cases.

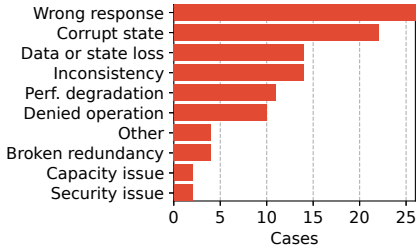


Figure 3: Consequence of the studied semantic failures.

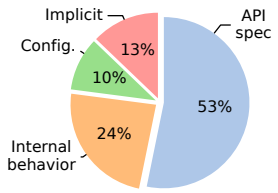


Figure 4: Sources of violated semantics.

systems' practices that may contribute to the differences.

Severity. How severe are these reported silent semantic failures? To answer this question, we analyze the severity levels that developers assign to the issues. Some systems use slightly different categories. We normalize them into three levels: *Blocker*, *Critical*, *Major*. Based on the official descriptions, *Blocker* means the issue "should block release until it is resolved"; *Critical* means the issue causes severe consequences like data loss; *Major* means a "major loss of function".

Overall, 45% of the studied cases have *Blocker* or *Critical* priorities. The ZooKeeper failure [13] described in Section 2.2 is an example *Blocker* issue. As another example *Blocker* issue, users reported that in their HDFS deployment, all the replicas of some blocks are residing on the same rack [8], which breaks the redundancy policy. This is clearly a severe violation because replica placement is critical to HDFS data.

We also compare the priority distribution of semantic failures with all failures in the sample. The result is shown in Figure 2. The average percentage of Blocker priority in semantic failures increases from 15% to 21%, and the percentage of Critical priority increases from 8% to 24%.

Interestingly, we find in some cases initially developers may not consider the symptoms to be severe, but after further investigation developers upgrade the priority level, *e.g.*, *"Marking as critical for 2.0. These 'unexpected behaviors' cause operator head-scratching and wasted hours of digging"* [5].

Finding 2: *Despite the lack of explicit error symptoms, silent semantic failures are considered severe by developers and users. Moreover, the sampled semantic failures are assigned with higher priorities compared to all sampled failures.*

Consequence. We next analyze the failure consequences. Figure 3 shows that besides incorrectness, semantic failures cause serious consequences such as corruption and data loss.

The consequences are damaging because clients or users are misled by the system's seemingly normal reactions. For example, Kafka guarantees that when a success response is sent to a producer, the produced message will be persisted by at least `min.isr` replicas. Otherwise, the producer will be notified of an error, so it may retry the request. In one failure [9], a leader replica switched to follower then back to leader. Some messages produced were lost while the client received responses with no error. This false success resulted in data loss for the users.

Note that Figure 3 is about the reported impact of failures, which is not always the semantic violation per se. For example, in a MongoDB case, the maximum cache usage configuration is not enforced. It takes a while for the violation to cause a performance problem—which is the consequence of this failure. But even before the system reaches the performance collapse, a cache limit violation has occurred.

Finding 3: *In addition to incorrectness (wrong responses), silent semantic violations often cause severe consequences including corrupt state, data or state loss, and security issues.*

5 What Kind of Semantics Is Violated?

5.1 Sources of Violated Semantics

The studied failures violate various system-specific semantics. We analyze where these semantics come from. There are four sources and Figure 4 shows their distributions:

- **API spec**: a system API promises certain effect will (not) occur, *e.g.*, a successful return of `removeWatch` API is supposed to remove the specified watcher.
- **Internal behavior**: the system's documentation explicitly guarantees that something should (not) occur about its *internal* behavior, which is not directly exposed to external APIs, *e.g.*, HDFS guarantees that if some Erasure Coding blocks fail, they should be detected and reconstructed.
- **User configuration**: user configurations regulate some system behaviors and the guarantees depend on the user settings. For example, the `max_hint_window_in_ms` parameter in Cassandra defines the maximum time window the coordinator will generate hints for a dead host.
- **Implicit**: the semantics are not explicitly defined or documented, but users expect them to hold for a correct system.

Finding 4: *Most (87%) studied failures violate semantics that are explicitly defined in API specs, system docs, or configs.*

Interestingly, in 10% of the studied cases, the system does not respect its configuration's semantics. For example, if users set `acl.inheritance` to `true`, HDFS should enable ACL inheritance; but in one case the inherited ACL permissions are masked [7]. This violation causes security issues. The problem of misconfiguration is extensively researched [20,21,35,56]. This finding suggests that even when users set configuration properly, a system can still misbehave.

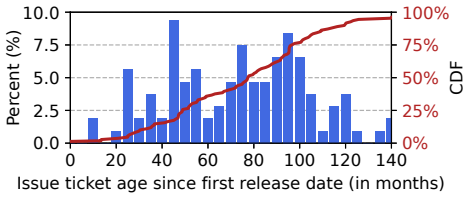


Figure 5: Issue ticket age (creation date minus first release date).

As an example of *Implicit* semantics, in one HBase case [4], a region is online in server A, but the region location registered in the meta table is server B. While this consistency semantics is a common sense, it is not explicitly declared.

Explicit documentation of semantics is indicative of developers' awareness of its guarantees and importance. One hypothesis is that if the semantics in a failure is not documented, it is understandable that developers did not make enough efforts to enforce the semantics. This finding disproves this hypothesis. However, the explicit documentations do *not* translate into fewer violations. One reason is that developers often document the semantics in a vague (*e.g.*, *"should produce correct results"*) or incomplete way. A more fundamental gap is that the documentation is designed to be human-readable but not machine-checkable. For example, the semantics for ephemeral `znode` in ZooKeeper is documented clearly, but the system does not have any mechanism or tool to enforce this semantics in deployment.

Implications: *Rich sources of documentation exist to leverage and judge semantic violations. Developers should move from documenting semantics in informal text to rigorously declare semantics that are mechanically checkable and enforceable.*

5.2 Categorizations of Violated Semantics

Old vs. New Semantics Modern distributed systems often keep adding new features. For example, the number of client APIs in ZooKeeper increased from 38 in version 3.4.6 (2016) to 78 in version 3.6.2 (2020). Similarly, HDFS' key APIs in `fs.FileSystem` increased from 128 in version 2.7.2 (2016) to 162 in version 2.10.0 (2019), along with significant increases of semantics in other interfaces such as RPC methods.

Since around 90% of our studied failures occurred after more than two years since the software's initial release (Figure 5), a natural hypothesis is that most of them violate some new semantics. We validate this hypothesis by analyzing the age of semantics in the studied failures. We define *old semantics* as ones that exist since the first major stable release of the system and others as *new semantics*.

Surprisingly, we find only less than one third (32%) of our studied failures violate relatively new semantics, while 68% of them violate old semantics. Old semantics usually represent the most fundamental functionalities the system provides since developers implement them first, and they usually undergo extensive testing already. However, our finding suggests that (1) even with new features added to the system, old semantics are still ones violated the most; (2) even with testing accumulating over the years, the reliability of old semantics is not necessarily higher in newer versions. Take ZooKeeper as an example. Its ephemeral `znode` interfaces and semantics have existed since the first major stable release (3.0.0) in October 2008 [1]. However, there are still production failures violating the guarantees of ephemeral `znode` reported by users even 10 years later [15].

We further investigate why old semantics still keep getting violated. There are three broad reasons: (1) *new implementation is buggy*, developers may optimize, refactor or refine the implementation of existing functionality, which contain bugs that break old semantics, *e.g.*, a concurrency bug introduced in changing an implementation to be multi-threaded; (2) *new feature adds buggy interactions*, when some new feature is added, developers may extend existing module to interact with or support the new feature. For example, after HDFS introduces the encryption zone feature, it needs to extend the original snapshot file function and the new handling path is buggy [6]; (3) *latent bugs are exposed*, as the most basic semantics, these old semantics' original implementations can be complex and contain latent bugs that can only be exposed in very specific scenario. In one ZooKeeper failure [14], users find the ephemeral `znodes` are not deleted when the system time changes unexpectedly. This bug exists for 6 years before it is discovered, because neither the testing nor most deployments would exercise the system with the time change.

Note that we did not count the numbers of semantics in the study, either for new or old semantics. This is because even with explicit documentation such as API specs, determining how many semantics are there for a given API can be subjective, which depends on the granularity of semantics. Instead, we objectively judge if the specific semantics violated in a failure were introduced in the initial release or not.

Finding 5: *68% of the studied failures violate old semantics.*

Implications: *Instead of having the false hope that old semantics are reliable, developers should invest efforts to prevent semantic violation regressions.*

Local vs. Distributed Semantics Since the study subjects are distributed systems, we analyze whether the semantic violations naturally require considering multiple distributed components. This question is important to the design of runtime verification techniques [43, 44, 48, 57].

We find that indeed 26% of the semantic violations require global information to judge, *e.g.*, whether the replica placement policy in HDFS is correctly enforced, or whether states in different Cassandra nodes match the consistency level.

However, interestingly, we find that the majority (74%) of the violations can be determined in a local scope. For example, `appendTo` in HDFS has the semantics of appending data to the end of a target file and making it persistent. A buggy node may fail to persist the new blocks or accidentally overwrite them. The violations can be determined in this node.

One reason is that a distributed system component often

keeps local copies of states for other components. For instance, even though ZooKeeper session is a global concept (a client connection to any follower or leader constitutes a session), such state is acknowledged to the ensemble. Thus, each node has a copy of the alive session and node list. The semantics of ephemeral `znode`, which require knowledge of the session information, can thus be checked locally in a ZooKeeper node.

Current runtime verification solutions typically aggregate global states across all nodes to check property violations. Obtaining such global information can be both expensive and tricky, *e.g.*, dealing with consistency issues in capturing distributed snapshots [23]. Our finding suggests it may be sufficient to use local checkers to expose many semantic failures.

Finding 6: *The violations in semantic failures can be usually (74%) determined in the scope of a single component.*

Implications: *Employing local checkers can potentially expose many semantic violations.*

Safety vs. Liveness Semantics Some failures break safety-related guarantees. For example, in Kafka, the maximum number of consumers in a group should not be larger than a configured limit, but users found more consumers joined the group.

In comparison, other semantics are liveness related. For example, ZooKeeper specifies that a container-type `znode` with no child `znodes` should *eventually* be deleted. Even when we observe some empty container node exists, it does not necessarily indicate this guarantee is violated because it might still hold some time later. Without context, one can interpret some safety guarantee, such as a correct response should be returned, to be involving liveness, because even if a response is not received, it could be still on the way. We refer to the system's official documentation for making the distinction. If the documentation explicitly states that when an operation returns, something (*e.g.*, a notification) will *eventually* happen, then a failure about its absence is a liveness violation.

It is generally challenging to check liveness properties [38], because there can be infinite possibilities in the execution that eventually produce the desired effect. Fortunately, we find most (86%) of our studied failures violate safety semantics.

Finding 7: *86% of the studied cases violate safety semantics.*

Implications: *There is usually a fixed time point to determine if a system has violated its semantics.*

6 Why Do Silent Semantic Failures Occur?

We analyze what causes a system to break its semantics. We are interested in identifying potential common bug patterns in the root causes, which can inform the designs of bug finding tools to eliminate semantic failures before production.

Some semantic failures are caused by bugs such as memory error, data race, and integer overflow, which are well studied with many tools designed to detect them. We find only 12% of the cases are caused by such bugs. The remaining failures are caused by system-specific logic bugs including design flaws, which are difficult to be caught by bug detection tools.

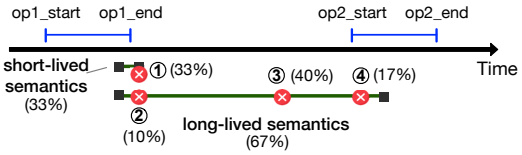


Figure 6: Timing of semantic violation.

An interesting finding is that even for failure cases that violate the same or related semantics, their root causes can be quite different. Take the ZooKeeper ephemeral `znode` as an example: (1) ZK-1208 is caused by a race condition: when ZooKeeper is handling the close session request, it deletes ephemeral `znodes` and then removes the session, in between a create operation causes new ephemeral `znodes` to be added; (2) In ZK-3144, the violations are caused by an incorrect order: during request processing, the `lastProcessedZxid` is updated before sessions are modified, so a snapshot may not include the change and the ephemeral node is not deleted after log replay; (3) In ZK-2355, the violations are caused by buggy error handling: follower fails while reading the proposal packet, but resetting `lastProcessedZxid` is missed in the error handler; (4) In ZK-2774, the system time of a server is changed unexpectedly, and session expiration codes rely the absolute system time, which causes the ephemeral `znodes` to persist after the client is disconnected for a long time.

Finding 8: *Only 12% of the studied failures are caused by well-defined bugs such as race conditions, while most cases are caused by a wide variety of logic bugs. Even for failures violating the same semantics, the root causes are diverse.*

Implications: *It can be challenging to exploit code patterns to eliminate semantic violations through static bug detection.*

7 How Are Semantic Failures Manifested?

Timing of Violation Understanding when semantics are violated can shed light on how to detect the violation.

As Figure 6 shows, some semantics only exist during the execution of its associated operation (at return point), *e.g.*, `read` operation should return the latest data. We call them *short-lived* semantics. In comparison, some semantics exist even after its associated operation finishes, *e.g.*, the specified file in `create` operation should be persisted and continue to be available after `create` returns. They often only cease to apply after some other event, *e.g.*, until a `delete` operation on the same file is executed. We call them *long-lived* semantics.

Interestingly, we find that 67% of the cases violate long-lived semantics. This is partly because these semantics have a larger "vulnerability" window compared to short-lived semantics: a violation can occur anytime in its lifespan. ZooKeeper ephemeral `znode` and watches are such examples. Essentially, the system must **maintain** the promise for a long time.

We categorize the violation timing into four scenarios: at the end of short-lived semantics (①), *e.g.*, wrong response, at the start (②) or in the middle (③) or near the end (④) for long-lived semantics. An example for ② is in HDFS-12217 the snapshot operation did not capture all open files, which

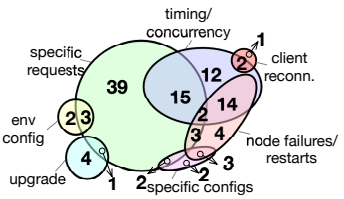


Figure 7: Distribution (# of cases) of the failure triggering conditions. Some combinations are omitted in the diagram for readability.

violates the long-lived snapshot semantics since the beginning. An example for ③ is HDFS-9083: at the block creation time, the block placement policy is honored; but after some node failures, all replicas of the block reside on the same rack. ④ happens when the semantics should cease to apply but did not, *e.g.*, ephemeral nodes should be removed when clients timeout. Figure 6 shows the distributions of the four scenarios.

Finding 9: *Near two thirds of the studied cases violate some long-lived semantics. In 40% of the cases, the semantics are initially honored but are violated in the middle.*

Implications: *It is crucial to* continuously *monitor semantic guarantees, even after the initial semantic check passes.*

Failure Triggering Conditions We further examine what triggers the semantic failures. Figure 7 shows the result.

Finding 10: *More than half of the studied failures are triggered by specific requests, while 39% of the failures require particular timing to trigger. Semantic failures often (41%) only manifest themselves under multiple types of conditions.*

HDFS-14514 is an example of semantic failure that requires multiple *types* of triggering conditions. The semantic violation (read out file size from snapshot is incorrect) can be only triggered when 1) `snapshot.capture.openfiles` is set; 2) create empty directory and encryption zone; 3) a client keeps a file open for write under the empty directory; 4) append several times; 5) perform a maintenance operation, snapshot.

We also find that in 23% of the cases, the triggering condition is certain system maintenance operation, such as compaction, cluster upgrade, node decommissioning. Such events do not occur frequently. They trigger semantic violations often because during the maintenance operation, the system execution enters a different mode, which exposes rare bugs.

Implications: *The reliability of semantics is vulnerable to maintenance operations or node events. Operators and the system should check violations during and after such actions.*

8 Current Practice for Semantic Failures

8.1 Testing

Since semantic violations concern functionality correctness, testing is responsible for catching them. The prevalence (Section 4) of many semantic failures in production seems to suggest a lack of testing. But that is not the case. The systems we study have extensive test cases—a median of 1309 test files. In addition, in 73% of the studied cases, the system has at least one test case covering the violated semantics.

Then *why the studied failures are not exposed during testing?* The earlier Finding 10 provides some clues. In many cases, even though there are related test cases, they lack some operations or arguments key to trigger the production failure. Even when the test cases have the proper operations and arguments, they only exercise the system under one timing, one configuration or normal scenarios, while the bugs are only triggered with unique timing, configuration, or node failures.

Are the failure triggering conditions so special that it is impossible for developers to foresee? Interestingly, we find that in many cases, similar triggering scenarios do exist in the test suite but they are not used in testing the violated feature.

Finding 11: *Semantic violations occur not simply due to a lack of testing. The violated semantics are usually (73%) covered by some existing test. In more than half of the studied failures, similar triggering conditions exist in the test suite.*

A fundamental gap is that developers tend to write tests driven by examples or fixes for a specific bug. Such tests are not expressive enough to preserve the underlying semantics and prevent regression. Consequently, developers spend repeated efforts to add tests. In HDFS-14514, the server reads snapshot file with incorrect length from encrypted zones. This exact semantics is already checked in an existing test case. If that test "copies" one line of test configuration `dfsAdmin.createEncryptionZone(...)` from other tests, the new bug will be triggered and exposed.

Implications: *Coverage of semantics alone is insufficient. Developers should introduce variances in existing test cases. It is also useful to "copy" triggering conditions across tests. More fundamentally, developers should write more general tests for the semantic properties rather than specific examples.*

8.2 Assertions

Assertions are a common method for catching logic bugs, which are major contributors to semantic failures (Section 6). They are typically only used in development and are turned off by default in release build for performance and stability.

Some of our studied systems use assertions in production: MongoDB has added many invariant checks since 2014 [11]. Interestingly, as Section 4 shows, MongoDB has the lowest ratio of semantic failures compared to other systems. While this practice may cause instability, *e.g.*, some users got infrequent crashes due to invariant check failures after upgrading to new versions [12], developers still prefer to fix the underlying bugs rather than turning off assertions completely.

We observe two gaps in the current practice. First, most existing assertions are pre-condition checks on the sanity of function arguments. They are too low-level to catch semantic violations, which require checking system functionalities and usually the operation history (*e.g.*, in checking consistency violations [48]). Second, existing assertions are usually only activated once during an operation, *e.g.*, the entry of a function. But many semantics are long-lived (Section 7), which require

continuous validation until the lifespan of semantics ends.

Finding 12: *Although in 51% of the failures the buggy functions have some sanity checks, few (9%) cases can be potentially detected by adding proper sanity checks.*

Implications: *Enabling assertions helps reduce silent semantic violations. However, developers should add more semantic-level invariant checks besides sanity checks.*

8.3 Observability

Since our studied failures are silent violations, *how do users notice these subtle failures then*? Understanding this question can reveal insights to improve the observability of semantic failures. We carefully examine the discussion threads in each ticket. In 34 cases, users mentioned their experience clearly.

For all of these cases, users discovered the issues through noticing something suspicious in some "side channels". We categorize them into two types: (i) benign errors in other requests (32%); (ii) anomalies in logs, files, or performance of other tasks (68%). In HBASE-11654, users find out the violations by noticing splitting directories in `/hbase/WALs/`, which is "very strange" because *"those logs should have been replayed and deleted"*. In KAFKA-9137, users observe the failure by seeing an increase in eviction rate in the logs. In CASSANDRA-6527, users found tombstones appeared even though they never used `delete` for a column family.

It might seem that we can rely on users to manually detect system semantic failures. Note that there is a survival bias: our studied cases by definition are identified, but in practice silent semantic violations can be easily missed because (i) users do not monitor the systems 24×7; (ii) when they check, they may not inspect the proper signals. When users notice the failures, the damage may be already done. In CASSANDRA-6527, users commented: "*Fortunately, we have noticed that quickly and canceled the migration. However, we were quite lucky.*"

How to make semantic failures more observable? First, if a system API has no interaction with others, it is hard to judge its correctness based on a single piece of information. In practice users often use multiple related APIs to cross-compare results. In HBASE-15236, users observe the violations because `Get` and `Scan` return different sizes for the same bulkloaded hfiles. Second, current systems often do not expose enough information about theirs internal states, thus users have to *ad-hocly* infer whether a promise is obeyed or not. Existing error messages (*e.g.*, a legitimate exception for another request) only focus on the current request, which is hard to link to the semantic violation in past correlated requests.

Finding 13: *Semantic violations are currently observed from "side channels": 32% from errors in other requests, 68% from anomalies in logs, files or performance of other tasks.*

Implications: *Designs of overlapping APIs improve observability of semantic violations. Systems should provide more admin APIs for convenient query of their internal states. Error messages should provide hints about past correlated requests.*

9 Oathkeeper: A Semantic Violation Checker

Guided by our study, we build a tool *Oathkeeper* to check semantic violations for large-scale distributed systems.

9.1 Design Overview and Workflow

Oathkeeper takes a runtime approach to check semantic violations in production. This choice is motivated by our findings that semantic failures have diverse root causes (Finding 8) and often difficult to expose in testing due to complex triggering conditions (Finding 10).

Central to a runtime verification approach is what invariants to use. Existing solutions rely on users to write distributed assertions to check the correctness of distributed protocols [43, 44] or network functions [57]. In those scenarios, the semantics to check are limited and well-defined. But in our cases, the systems have abundant (Table 1) and loosely-defined semantics. Even for semantics that can be described in simple expressions informally, mapping them to the concrete checkable invariants in the complex systems code is hard. These factors make manual construction a daunting task.

Insight and Key Idea. The insight behind Oathkeeper is based on our finding that the majority of the studied failures violate old semantics (Finding 5) despite the decent coverage of testing (Finding 10). When a semantic failure occurs, developers usually add regression tests. But these tests only check if the specific bug is fixed in a specific setup, while the same semantics can be violated repeatedly in other scenarios.

Based on this insight, Oathkeeper leverages the existing regression tests developers write for past semantic failures and automatically extracts the essence—the violated semantic rules. Oathkeeper then enforces these rules at runtime to detect future semantic violations, which may be caused by different bugs under different conditions.

Input and Output. To apply Oathkeeper to a new system, users supply a system-wide configuration and a list of past semantic failure metadata. The former provides basic information about the system such as the compilation command and test directory, and optionally the classes to include for analysis. The latter metadata is provided in the form of git commit id (for version switching) and regression test name.

Oathkeeper outputs the likely semantic rules (Section 9.3). Prior runtime verification tools focus on invariants expressed as predicates among key state variables in a system such as `lock_id` and `lock_mode`. This representation alone can be insufficient or complex to express the semantics of large distributed systems. Instead, Oathkeeper focuses on rules that describe relations among semantics-related events, particularly operation invocations and state updates. Such an *event relation* rule is expressive to capture various semantics.

Workflow. Figure 8 shows the tool's workflow. Oathkeeper operates in two stages. In the offline stage, Oathkeeper instru-

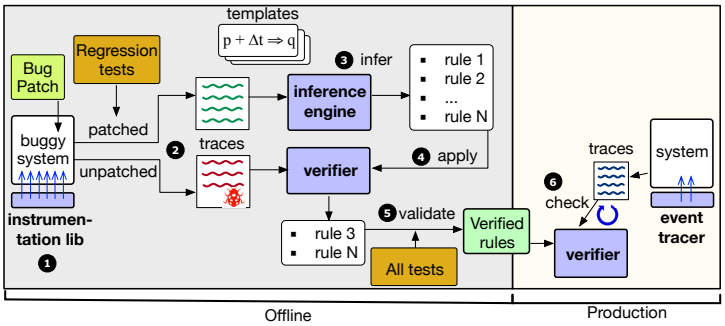


Figure 8: Workflow of Oathkeeper.

ments the target systems to record major events (❶). It then exercises the system twice with the regression tests: once using the patched version and the second time using the buggy version. This will generate two sets of traces (❷). The *inference engine* infers likely semantic rules from the traces of the patched version (❸). The *verifier* applies the inferred semantic rules against the traces of the buggy version and output rules that are violated in the buggy traces (❹). We assume these violated rules are potentially related to the semantic failure. Further optimizations are applied to remove noises and redundancies (❺). In the online stage, Oathkeeper only performs minimal instrumentation that is relevant to these final semantic rules from the offline stage. The event tracer ingests traces from the system in real time. The Oathkeeper verifier continuously checks the traces against the deployed semantic rules and reports violations (❻).

9.2 Instrumentation and Trace Generation

For both inferring semantic rules and runtime verification, we need to first instrument the system to obtain execution traces. The Oathkeeper traces use a uniform *event* schema that captures operation-related events and state-related events.

Oathkeeper designs a *load-time instrumentation library* that performs bytecode manipulation when a target system is loaded. This way of instrumentation is convenient (without re-compiling and re-packaging the system) and transparent.

To record operation events, the library adds hooks at the beginning, return and exception point of a method. To record state events, Oathkeeper takes a patch plus base approach. It analyzes the given semantic failure patch and automatically includes the list of classes involved in the patch file. Users can optionally specify names of some important system classes, such as `SessionTrackerImpl`. With the combined list of classes, Oathkeeper performs simple analysis at the loading phase of these classes to retrieve their member variables of primitive or collections types, and treat them as the state variables. It then identifies instructions that *update* these variables and insert a hook to emit a state update event with the relevant context (variable name, location, *etc.*).

For each given test, Oathkeeper switches the target system to the patched version. The tool executes the test with the instrumented system and generates the trace of events.

Template	Example
$p \Rightarrow q$	decommission a datanode should trigger reconstruction
$s\uparrow \Rightarrow p$	when datanode changes, associated watcher notifies clients
$s\uparrow \Rightarrow k\uparrow$	after session disconnection, ephemeral node is removed
$(s = c) \oplus q$	read-only server should not provide write access
$p + \Delta t \Rightarrow q$	inserted data should expire after the TTL is reached.
$s\uparrow \rightarrow q$	cf schema should be altered before alter command returns
$p \Rightarrow \odot(s\uparrow, k\uparrow)$	after snapshot renaming, either new snapshot creation and old snapshot deletion both occur or none of them occur

Table 3: Some templates integrated in Oathkeeper. p, q are operations, s and k are states, t is time, c is constant. $\neg p$ means p can not occur. $\uparrow$ means state changes. $p + \Delta t$ means time t after p occurs.

Then Oathkeeper reverts the target system to the buggy version (snapshot prior to the patch commit id). Since the buggy version does not contain the test, Oathkeeper copies the regression test from the patched version and executes it to get the buggy trace. If the test cannot directly run on the buggy version due to interface changes (*e.g.*, a function used in the test is not public in the buggy version), the tool supports user-provided patches to fix the compatibility issue.

The trace is stored in a JSON file for ease of deserialization. An example trace entry is `{"type": "OpTriggerEvent", "data":{"opName": "zookeeper.FileSnap.deserialize", "time": 1654026992, ...}}`. The trace scale is usually moderate, because it is generated from tests. For example, with ZooKeeper, even under the full instrumentation mode (instrumenting all classes), most end-to-end tests generate less than 10,000 events. A common scale is several thousands. We see large traces in only 5/273 tests that produce over 500,000 events. Under the diff mode (only instrument the classes affected by the patch), the trace typically has hundreds of events.

9.3 Template-Driven Inference

A key challenge in the semantic rule inference step of Oathkeeper is to integrate domain knowledge without requiring significant manual effort, while also having reasonable accuracy and efficiency. We take a template-driven approach to address this challenge. We first summarize general semantic rule patterns, such as happens-before relationship, atomicity, periodicity. For each pattern, we define one or more parameterized templates, such as a state change event for s must happen before the completion event of operation p.

```java
public abstract class InferScanner {
  //init state variables
  abstract void prescan(Set<Event> eventSet);
  //always need to go through the whole traces
  abstract void scan(Event event);
  //check states after scan, and generate invariants
  abstract List<Invariant> postscan();
}
public abstract class VerifyScanner {
  //init state variables
  abstract void prescan();
  //return true if continues to scan, otherwise break
  abstract boolean scan(Event event, Context context);
  //check states after scan, and judge
  abstract InvState postscan();
}
```

Listing 1: Inference and validation interfaces for each template.

Algorithm 1: Generic inference and validation workflow.

Input: L: a trace (list of events)
Output: a list of inferred invariants (one inv. is a template w/ context)
Func `Infer`(L)**:**
    `/* get unique events in the trace (we define equality individually for different types of events) */`
    `unique_events` ← Set(L)
    `prescan(unique_events)`
    foreach *event* ∈ L **do** `scan(event)`
    return `postscan()`

Input: L: a trace (list of events), *context*: parameters in templates, *e.g.*, if an invariant is $a1 \Rightarrow a2$, context is a1 and a2
Output: the checking result of invariant (pass, fail or inactive)
Func `Verify`(L, *context*)**:**
    foreach *event* ∈ L **do**
        if *scan(event,context)* **then break**
    return `postscan()`

Func `Main`($L_{patched}$, L_{buggy})**:**
    `inv_list` ← ∅
    foreach *inv* ∈ *Infer*($L_{patched}$) **do**
        if *Verify*(L_{buggy}, *inv.context*) == *InvState.FAIL* **then**
            `inv_list.add(inv)`
    return `inv_list`

Oathkeeper currently supports 18 templates. Table 3 shows several examples. Our technical report [47] shows the full list.

The inference engine implements an inference algorithm for each template. The algorithm checks if there are matches in a given trace and derives concrete values to each template parameter if so. We call each match a *context* for the template, which is a potential invariant. For one template (*e.g.*, $p \Rightarrow q$), a trace can have multiple contexts (*e.g.*, $a1 \Rightarrow a2$ and $a1 \Rightarrow a3$).

The templates allow encoding domain-specific semantics without significant specification effort. They also restrict the search space so the inference engine only analyzes trace events that match the template structure and parameter types. While these templates may not represent the exact or full semantics like a high-level specification does, they can capture the essential ingredients for making the semantics hold.

The inference engine takes the trace obtained from running the regression tests against the patched system. Each template class implements an `infer` function that returns a list of

Algorithm 2: Implementation for template $p \Rightarrow q$.

Func `ImplyTemplate::InferScanner::prescan`(S)**:**
    foreach *event* ∈ S **do** `C.put(event, {})`
    foreach *event* ∈ S **do**
        foreach *event2* ∈ S **do**
            if *event != event2* **then**
                `C.get(event).put(event2, 0)`
                `C.get(event2).put(event, 0)`
Func `ImplyTemplate::InferScanner::scan`(*event*)**:**
    foreach *(k,v)* ∈ *C.get(event)* **do** `v` ← `v + 1`
    foreach *event2* ∈ *C* **do**
        if *event* == *event2* **then continue**
        `val` ← `C.get(event2).get(event)`
        if *val* > *0* **then** `C.get(event2).put(event, val - 1)`
Func `ImplyTemplate::InferScanner::postscan`(L)**:**
    `lst` ← `[]`
    foreach *(k,v)* ∈ *C* **do**
        foreach *(k2,v2)* ∈ *v* **do**
            `/* add potential invariants when counter is 0 */`
            if *v2==0* **then** `lst.add(genImplyInv(k,k2))`
    return `lst`

Func `ImplyTemplate::VerifyScanner::prescan`()**:**
    `ifHold` ← `true`
    `ifActivated` ← `false`
    `counter` ← `0`
Func `ImplyTemplate::VerifyScanner::scan`(*event*, *context*)**:**
    if *event* == *context.left* **then**
        `counter` ← `counter + 1`
        `ifActivated` ← `true`
    else if *event* == *context.right && counter* > *0* **then**
        `counter` ← `counter - 1`
    return `true`
Func `ImplyTemplate::VerifyScanner::postscan`(L)**:**
    if *counter != 0* **then** `ifHold` ← `false`
    if *!ifHold* **then return** `InvState.FAIL`
    if *ifActivated* **then return** `InvState.PASS`
    else return `InvState.INACTIVE`

rules from the trace. Most templates follow three phases in the `infer` function: *pre-scan*, *scan*, and *post-scan* (interfaces defined in Listing 1). The pre-scan step typically builds an index of the unique event set in the trace. The uniqueness is determined by a custom function we define for different types of events. For example, operation invocation events are unique based on the signatures of invoked functions. The scan step iterates through each event in the trace and updates bookkeeping data structures such as an event occurrence map. The post-scan step generates invariants based on the bookkeeping data structures. Templates that do not follow this pattern can customize the procedures. For example, the `AfterOpAtomicStateUpdateTemplate` iterates forward once and scans backwards once; the `StateEqualsDenyOpTemplate` scans the trace for each state type in the test.

The core inference algorithm for each template, while different, is relatively straightforward. It essentially involves identifying events in the trace that match the type of a template's parameter, enumerating hypotheses (candidates) from the contexts, and validating the hypotheses against the trace. Since the trace size is moderate, we can afford enumerations.

Example. We describe the inference of a representative tem-

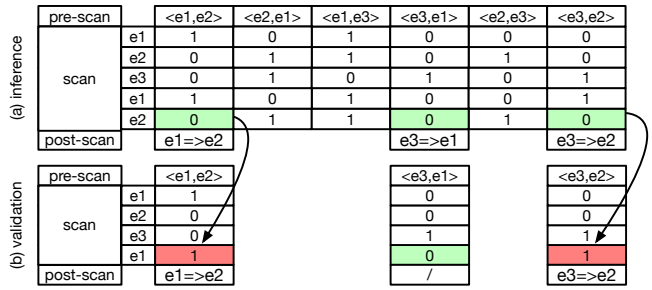


Figure 9: Inference and validation algorithm example.

plate $p \Rightarrow q$, which represents that every invocation of operation p implies a subsequent operation invocation of q. For example, `createSession` should usually imply `closeSession`. The steps are listed in Algorithms 1 and 2.

We use Figure 9 (a) to show the process of inferring rules of template $p \Rightarrow q$ from a patched trace $[e1, e2, e3, e1, e2]$. The algorithm assumes all pairs $<e_i, e_j>$ in the unique event set are candidate contexts to the template, in which e_i and e_j are of `OpTriggerEvent` type and the uniqueness is based on the operation name. Then it attempts to find counterexamples to invalidate wrong rules. The inference algorithm of this template uses a simple counting approach that runs in three steps. The *pre-scan* step constructs a nested map {`event`: {`event`: `int`}} to record the occurrences for the event pairs. For each event pair, the counter is initially zero. Then the *scan* step iterates through each event e in the trace in order. If e is e_i, *i.e.*, a key in the nested map, we increment the counters for all entries with $<e_i, *>$ keys; if e is e_j, we decrement the counters for entries that have $<*, e_j>$ keys *and* have positive counters. In the *post-scan* step, we check the final state of counters. If the final counter does not reach zero, there is an orphan e_i that does not have subsequent e_j. We get $e1 \Rightarrow e2$, $e3 \Rightarrow e1$ and $e3 \Rightarrow e2$ at the end. Rules like $e1 \Rightarrow e3$ are removed because no subsequent $e3$ occurs after the second $e1$.

9.4 Rule Validation

After step ❸, the inference engine could infer many likely semantic rules. Oathkeeper then applies these rules against the buggy traces (❹) and sees which rules are violated. Similarly to inference, each template class needs to implement a `verify` function. The `verify` function also usually consists of three phases: *pre-scan*, *scan*, and *post-scan*. The pre-scan step initializes auxiliary data structures specific to the template. The scan step goes over the events in the trace and updates the data structures. In some template, the scan step does not need to iterate through all events in the trace if a contradictory example is already found. The post-scan step checks the data structures and returns the result, which could be `PASS` (rule is activated and no contradiction is found), `INACTIVE` (the antecedent of the rule does not occur, *e.g.*, $p \Rightarrow q$ is inactive in a trace without occurrences of p), or `FAIL` (at least one contradiction is found). We only preserve rules that pass in the patched trace and fail in the buggy trace.

Example. Algorithms 1 and 2 show the steps to verify template $p \Rightarrow q$. We use Figure 9 (b) to show the process of validating inferred rules from (a) on a buggy trace $[e1, e2, e3, e1]$. There are three rules to verify: $e1 \Rightarrow e2$, $e3 \Rightarrow e1$, $e3 \Rightarrow e2$. In the *pre-scan* step, we first initialize a counter for each inferred rule. The *scan* step then updates the counter: for rule $e_i \Rightarrow e_j$, if a processed event e matches e_i, we increment the counter; if e matches e_j, we decrement the counter if it is positive. All three rules are active as both $e1$ and $e3$ appear in the trace. The *post-scan* step marks rules with non-zero counters as `FAIL`: $e1 \Rightarrow e2$ and $e3 \Rightarrow e2$.

However, there could still be a significant number of rules due to noises like unfinished tests (*e.g.*, an assertion failed in the middle of the test), new type events (new methods introduced), coincidence, and methods that are used for testing only. To reduce these noises, the verifier validates (❺) the candidate rules against traces obtained from all test cases, under the patched version, and *discards* rules that do not hold in all traces. In addition, we filter uninteresting rules about the system start-up or shut-down methods or thread run methods. This is achieved by inserting special marker events at the start and end of test method, and only running the inference algorithms on trace region within the markers.

9.5 Runtime Checking

Oathkeeper deploys the refined semantic rules with the target system in production, along with the verifier and event tracer. Oathkeeper performs load-time instrumentation to the production system in a wrapper class of the entry points. Different from the offline stage, the instrumentation is selective to only the deployed rules and is thus lightweight. The event tracer stores in-memory traces from the target systems.

The runtime verifier schedules periodical tasks that validate the current trace against *each* of the deployed semantic rules. It reuses the same checking logic defined in the function `verify` of the template. When the engine finds a semantic rule reported as `FAIL`, it records the counterexamples in the traces for debugging. It also schedules a second check on this violated rule again shortly to tolerate transient violations or inconsistencies in the trace. For high availability, Oathkeeper generates alerts in the log upon detection of potential semantic failures and does not attempt to crash the system.

9.6 Optimizations

The validation step can be time-consuming. With N (often thousands) candidate rules and M (often hundreds) test cases, we need to get M traces and check $N \times M$ times. To reduce the validation time, we introduce a survivor optimization. After a test finishes, we validate the rules, if some rule is already "killed" (invalidated) by this test's trace, it will not be carried over to the remaining tests. Therefore, only the survived rules will be validated to the end. Another optimization is to run more closely related tests first. The rationale is that some test takes a long time to run but is irrelevant to a given rule (thus the test's trace will not disprove the rule). By prioritization, we can potentially invalidate false rules faster.

We also add several optimizations to reduce the runtime overhead. First, the event tracer only preserves the most recent events within a time window, since always checking full traces from the start is wasteful. The time window is configured larger than checking frequency to avoid missing checking events. The events involved in time-related semantic rules are excluded as their expiration time is based on their parameters. Second, to achieve both high concurrency and low memory pressure, we decouple the checking from the event emission with a ring buffer design inspired by high-performance message queues [10]. Third, to avoid massive new object creation frequently triggering garbage collection, we reuse expired event objects in the ring buffers. Oathkeeper also pre-allocates buffers for each type of events at the instrumentation phase to prevent buffer initialization blocking.

9.7 Implementation

We implement Oathkeeper in Java (JDK 8). Its instrumentation library is built based on Javassist for class bytecode manipulation. Its test engine leverages JUnit to manage and execute test cases. The tool also includes a workflow script such as checking out patched and buggy versions and checking a semantic rule against given traces.

9.8 Limitations

Our approach makes several assumptions: 1) semantics should be expressible with simple relations of events; 2) the system has a number of test cases with good quality; 3) the failure patch should not involve significant redesign or interface interfaces. If some assumption does not hold, Oathkeeper may fail to deduce good semantic rules.

10 Evaluation

We have integrated Oathkeeper with ZooKeeper, HDFS and Kafka. We evaluate (1) whether Oathkeeper can leverage past semantic failures to check new violations; (2) what runtime overhead it incurs to the target system. The experiments are done in servers with 20-core 2.2 GHz CPUs, 64 GB memory, running Ubuntu 18.04. The Oathkeeper check engine is configured to schedule and check rule violations every second.

10.1 Generation Overview

Oathkeeper requires old semantic failures and their associated regression tests as input to extract semantic rules. We select old semantic failures and their regression tests to reproduce (8 for ZooKeeper, 10 for HDFS and 8 for Kafka). These tests cover major functionalities of the three systems. We add a switch in the system code to easily enable and disable the patch for the semantic failure bugs. We then apply Oathkeeper to the source code to add instrumentation points, run the regression tests with the patch switch turned on and off, and execute other steps in Oathkeeper (Section 9.1). For each case, Oathkeeper infers many raw semantic rules. After the validation and optimization step, the rule set is significantly reduced. In total, Oathkeeper extracted 285 rules for ZooKeeper, 1,209 rules for HDFS, and 150 rules for Kafka.

JIRA Id	Violated Semantics
ZK-1496	ephemeral node should be deleted after session expired
ZK-1667	watcher should return correct event when client reconnected
ZK-3546	container node should be deleted after children all removed
HDFS-14699	failed block need to be reconstructed
HDFS-14317	edit log rolling should be activated periodically
HDFS-14633	file rename should respect storageType quota
KAFKA-12426	partition topic ID should be persisted into metadata file

Table 4: Evaluated newer semantic failures.

10.2 Checking Newer Violations

We evaluate whether the inferred rules are useful to catch new semantic failures. Given Oathkeeper's approach, it is likely less effective with unseen semantics. We reproduce 7 newer (9–34 months later) failures (Table 4) that violate related semantics in the old cases, but with different root causes. With the inferred rules, Oathkeeper detects violations for 6 of them. These newer violations are known bugs by the time we conducted this experiment. However, their root causes and triggering conditions are completely different from the failures used to extract semantic rules. Oathkeeper detects these newer violations with only knowledge from the old failures, which demonstrates the tool's detection capability.

We show one example in Figure 10. ZK-1496 is not in our study dataset, but its symptom is similar to a studied failure ZK-1208 that was reported 9 months ago prior to ZK-1496 in an older release. Users found that the ephemeral `znodes` were not deleted long after the client exited. The root cause is a race condition bug that while the session tracker is removing the expired session, another thread is processing an ephemeral node creation request. In ZK-1208, developers added a fix to mark sessions as closing to prevent ephemeral node creation on expiring sessions, and introduced a regression test. Oathkeeper executes the regression test on ZooKeeper twice with patch enabled and disable, and generates two traces (c) and (d). Then Oathkeeper infers rules (e) from the patched traces. Not all inferred rules are useful. Oathkeeper only preserves rules that fail in buggy traces and pass all tests (f). Rules such as ③ are filtered when being validated on all tests. Finally, two verified rules ① and ② detect the violations (g).

Oathkeeper fails to detect ZK-1667: client A sets a watch on `/d` and then disconnects, client B deletes `/d` and recreates it; when client A reconnects, it receives a `NodeCreated` event instead of `NodeDataChanged` event. The violated semantics fits into one of our templates. However, due to the quality of the old watch test in our pool, Oathkeeper infers other rules.

The average detection time is 0.91 seconds. This result does not contradict with the long-lived semantics finding in Section 7. In the experiments, we trigger the conditions to reproduce the failure soon and measure the detection time from the start time of the violation.

We compare Oathkeeper with a state-of-the-art invariant

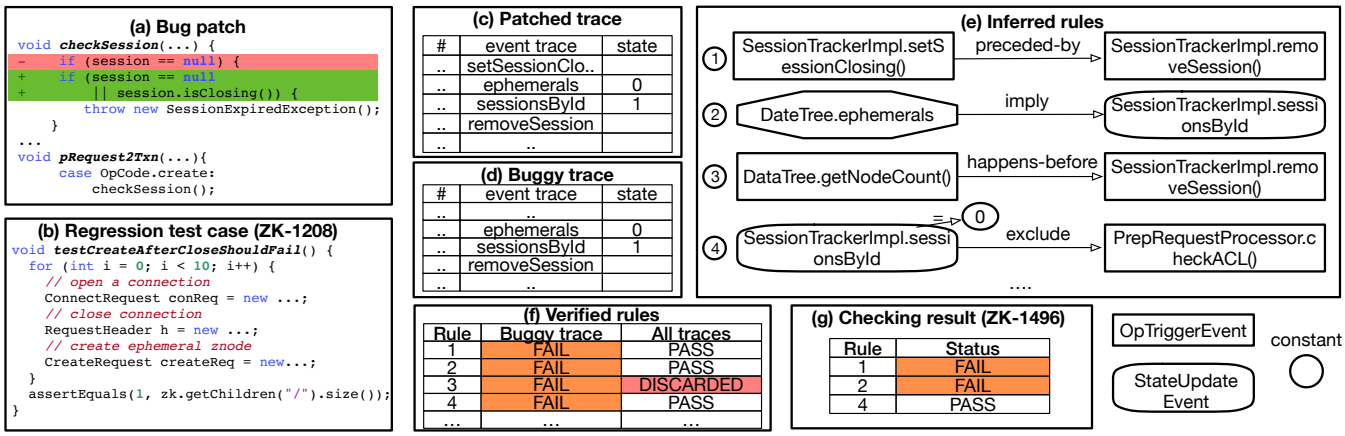


Figure 10: Example: Oathkeeper workflow of using ZK-1208 to detect ZK-1496.

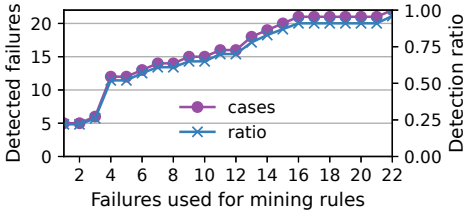


Figure 11: Detection of 22 semantic failures in ZooKeeper (sorted by the bug ticket time in ascending order) when applying Oathkeeper on a sliding subset of the failures for inferring semantic rules.

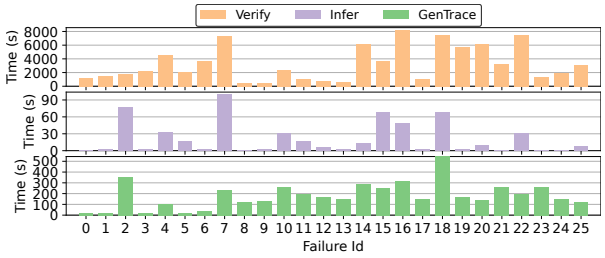


Figure 12: Time to generate trace, infer rules, and verify rules against test suite. ZK: Id 0–7, HDFS: Id 8–17, KF: Id 18–25.

checking tool, Dinv [30]. Dinv is designed for checking distributed protocols. Its core invariant inference component is based on Daikon [26] that mines variable-level relationship. We instrument the state variables in the two systems and apply Daikon to traces from the system test cases. The inferred invariants only detect 1 case (ZK-1496) and are highly noisy.

We conduct an additional "cross-validation" experiment. Specifically, we collect a larger pool of 22 semantic failures in ZooKeeper. The failures are sorted from older to newer. We feed each failure to Oathkeeper and measure how many of the 22 failures can be detected. For 16 cases, the rules inferred from one case only detect that case. It does not imply, though, these rules are useless. They might help detect failures outside the pool. Interestingly, for the remaining 6 cases, their inferred rules detect a median of 5 failures. For example, rules from ZK-2355 can detect 6 other failures besides itself. Figure 11 plots the aggregate detection result.

10.3 Performance

Figure 12 shows the performance of running Oathkeeper for the 26 old cases. Our template-based inference is fast. The median time to finish inference is 6.5 s. The median trace generation time is 153.5 s. The most time-consuming part is verifying the inferred rules against the system test suite, because running the full test for the three systems alone takes a long time. The end-to-end validation time is 2196 s (median). After discounting the original test execution time, the median validation time is 301 s. The survivor optimization we introduce (Section 9.4) helps. In one time-consuming case, it reduces the end-to-end validation time from 8104 s to 5024 s.

	Base	25%	50%	75%	100%
ZooKeeper	418.27	417.63	416.71	416.55	416.1
HDFS	174.55	174.56	172.10	172.10	172.06
Kafka	30,759.49	30,546.00	30,377.50	30,246.04	30,183.15

Table 5: System throughput (op/s) with varying percentages of semantic rules enabled. The 100% represents 285 rules for ZooKeeper, 1,209 rules for HDFS, and 150 rules for Kafka.

10.4 Runtime Overhead

We measure the overhead Oathkeeper introduces to the systems at runtime. The main source of overhead comes from the added instrumentation to emit traces; the rule checking does not impact the system much because it is done asynchronously. Oathkeeper only adds instrumentation relevant to the deployed rules to minimize the overhead. Naturally, more rules lead to higher overhead. We evaluate the overhead as a function of the percentage of enabled rules. For ZooKeeper, we run the workload of 15 clients sending 15,000 requests (40% reads, 60% creates and writes). For HDFS, we run the built-in benchmark NNBenchWithoutMR which creates and writes 100 files, each file has 160 blocks and each block is 1MB. For Kafka, we run the workload of producing 1 million 16KB messages. Table 5 shows the result. With all rules enabled, the average system throughput overhead is 1.27%.

Our initial event tracer used an array list with synchronization, which resulted in a 31% overhead under heavy workloads. We later implemented a more complex non-blocking queue, but the overhead is still large. After investigation, we

found the overhead mainly comes from memory and GC instead of synchronization, which motivated our ring buffer design (Section 9.6) that significantly reduced the overhead.

10.5 Rule Activation and False Positive

We deploy the inferred rules to a cluster of ZooKeeper, HDFS, and Kafka instances. We run a set of workloads against the instances. We first measure the rule activation ratio during the experiment. A rule is activated if the check engine finds the antecedent of the rule has occurred. For ZooKeeper, 11% of the rules are activated. The remaining rules are not activated due to the lack of workloads, faulty conditions, *etc.*, to trigger the antecedent events. For HDFS and Kafka, the activation ratio is 66% and 48%. We then measure the false positive ratio among the activated rules. The result is 4% for ZooKeeper, 9% for HDFS, and 12% for Kafka. This result benefits from the validation steps described in Section 9.4: Oathkeeper eliminates falsely inferred rules by validating the rules against both the buggy trace and the traces from all test cases of a target system. Adding profile runs or a dynamic ban mechanism can further remove the false rules.

11 Related Work

Semantic Bugs. Several studies [42,55,58] analyze the prevalence of semantic bugs in open-source server software. Our study analyzes *semantic failures* in distributed systems. Beyond the difference that we investigate distributed systems, the study of *bugs* is in general a complementary effort to the study of *failures*. The former focuses on analyzing the static code patterns, while the latter focuses on the dynamic manifestations and system misbehavior.

Several solutions are proposed to detect semantic bugs in file systems and DBMS, including cross-checking multiple file system implementations [50], fuzzing [39], and testing using pivoted query [54]. Both cross-checking and fuzzing focus on finding bugs offline. Oathkeeper focuses on a complementary direction of inferring semantic rules for runtime checking. We hope our study can motivate future work to extend these solutions to detect semantic bugs in distributed systems. We observe some open challenges to cross-check distributed systems: distributed systems usually provide a wide variety of semantics that are less rigorously specified compared to file systems, which have well-defined semantics (*e.g.*, POSIX standard) and many implementations. Each distributed system has its unique semantics and may not be cross-checkable. In addition, they often contain many internal and background mechanisms that provide semantic guarantees but the semantics are not easy to be tested. For fuzzing, the challenge is that many silent semantic violations require external faulty events (*e.g.*, node restarts, network error) to trigger besides input. Thus, fault injection testing is needed.

Distributed Systems Failure Study. Understanding failures has been an important theme in distributed system literature, with a series of empirical studies [16, 17, 19, 25, 31, 32, 36, 37, 46, 51], *e.g.*, on fail-slow faults [32], gray failures [37], and network partitions [17]. These failures usually have some error signals such as timeouts. Our study complements these studies and focuses on the under-explored silent semantic failures in distributed systems.

Runtime Verification. Prior works have explored runtime assertions to verify distributed protocols [43, 44], file systems [28], and network functions [57]. Runtime verification [34] is also studied in embedded systems and Java benchmark programs [24]. Recent works [45, 46] propose intrinsic watchdogs that detect partial faults with clear error signals. Lu *et al.* propose a runtime checker for consistency violations [48]. Overall, there is a lack of runtime verification solutions for monitoring the semantic correctness of large-scale distributed system implementations. Our proposed tool Oathkeeper explores automatically extracting semantic rules to check a variety of semantics for large distributed systems.

Invariant Mining. Inferring likely invariants from software execution traces have been studied, *e.g.*, Daikon [26] and DIDUCE [33]. They mainly focus on mining invariants on the relationship of program variables for single-component software, *e.g.*, `off < array.length`. These invariants are too low-level to capture the semantics of distributed systems.

Dinv [30] is proposed to infer protocol invariants of program variables across nodes. It runs complex program slicing to instrument program variables influenced by network communication. It then uses Daikon to infer invariants from the logs of running the system's test suite. I4 [49] infers inductive invariants for verifying distributed protocols.

Oathkeeper is complementary to the two efforts. Instead of protocols and variable relations, we focus on inferring high-level semantic rules for large distributed systems, most of which are not about protocols. Also unlike Dinv, Oathkeeper does not rely on complex static analysis to work and thus does not suffer from analysis inaccuracies and scalability limitations. Oathkeeper takes a unique approach of leveraging past failures and semantic templates to extract semantic rules.

12 Conclusion

Silent semantic violations pose a severe challenge to distributed systems reliability. This paper sheds light on this under-explored yet important problem by presenting a study on real-world failures in popular distributed systems. It reveals that sadly "a promise is often *not* a promise". Guided by our study, we design a tool Oathkeeper that automatically extracts semantic rules from past semantic failures, and enforces these rules at runtime to check future violations.

Acknowledgments

We thank our shepherd, Annette Bieniusa, and the OSDI reviewers for their valuable feedback. This work was supported in part by NSF grants CNS-1942794, CNS-2149664, CNS-1910133, and CCF-1918757, and a Facebook research award.

References

[1] Apache ZooKeeper releases. https://zookeeper.apache.org/releases.html.

[2] Google cloud infrastructure incident #20013. https://status.cloud.google.com/incident/zall/20013.

[3] Google cloud storage incident #17005. https://status.cloud.google.com/incident/storage/17005.

[4] HBASE-11536: Puts of region location to meta may be out of order which causes inconsistent of region location. https://issues.apache.org/jira/browse/HBASE-11536.

[5] HBase-17125: Inconsistent result when use filter to read data. https://issues.apache.org/jira/browse/HBASE-17125.

[6] HDFS-12217: HDFS snapshots doesn't capture all open files when one of the open files is deleted. https://issues.apache.org/jira/browse/HDFS-12217.

[7] HDFS-14359: Inherited acl permissions masked when parent directory does not exist (mkdir -p). https://issues.apache.org/jira/browse/HDFS-14359.

[8] HDFS-9083: Replication violates block placement policy. https://issues.apache.org/jira/browse/HDFS-12070.

[9] KAFKA-2960: DelayedProduce may cause message loss during repeated leader change. https://issues.apache.org/jira/browse/KAFKA-2960.

[10] Lmax disruptor. https://lmax-exchange.github.io/disruptor/.

[11] MongoDB-12355: add "invariant" for invariant checking in server code. https://jira.mongodb.org/browse/SERVER-12355.

[12] MongoDB-50971: Invariant failure, wt_notfound: item not found. https://jira.mongodb.org/browse/SERVER-50971.

[13] ZooKeeper-1208: Ephemeral node not removed after the client session is long gone. https://issues.apache.org/jira/browse/ZOOKEEPER-1208.

[14] ZooKeeper-2774: Ephemeral znode will not be removed when sesstion timeout, if the system time of zookeeper node changes unexpectedly. https://issues.apache.org/jira/browse/ZOOKEEPER-2774.

[15] ZooKeeper-3144: Potential ephemeral nodes inconsistent due to global session inconsistent with fuzzy snapshot. https://issues.apache.org/jira/browse/ZOOKEEPER-3144.

[16] M. Alfatafta, B. Alkhatib, A. Alquraan, and S. Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 351–368. USENIX Association, Nov. 2020.

[17] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 51–68, Carlsbad, CA, USA, 2018.

[18] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[19] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HotOS '01, pages 33–. IEEE Computer Society, 2001.

[20] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 307–320, 2012.

[21] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–11, 2010.

[22] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[23] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[24] F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, page 569–588, Montreal, Quebec, Canada, 2007.

[25] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 14:1–14:14, Santa Clara, California, 2013.

[26] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, Dec. 2007.

[27] A. S. Foundation. HDFS high availability using the quorum journal manager. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html.

[28] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 7–7, San Jose, CA, 2012.

[29] J. A. Goguen. Semantics of computation. In *Proceedings of the Proceedings of the First International Symposium on Category Theory Applied to Computation and Control*, page 151–163. Springer-Verlag, 1974.

[30] S. Grant, H. Cech, and I. Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1149–1159, Gothenburg, Sweden, 2018.

[31] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, SOCC '16, pages 1–16, Santa Clara, CA, USA, Oct. 2016.

[32] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 1–14, Oakland, CA, USA, 2018.

[33] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 291–301, Orlando, Florida, 2002.

[34] K. Havelund and G. Roşu. Runtime verification. *Computer Aided Verification (CAV '01) satellite workshop (ENTCS)*, 55, 2001.

[35] Y. Hu, G. Huang, and P. Huang. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, OSDI '20. USENIX, November 2020.

[36] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[37] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17. ACM, May 2017.

[38] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI '07, page 18, Cambridge, MA, 2007.

[39] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147–161, Huntsville, Ontario, Canada, 2019.

[40] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, Cascais, Portugal, Oct. 2011.

[41] S. Levy, R. Yao, Y. Wu, Y. Dang, P. Huang, Z. Mu, P. Zhao, T. Ramani, N. Govindraju, X. Li, Q. Lin, G. L. Shafriri, and M. Chintalapati. Predictive and adaptive failure mitigation to avert production cloud VM interruptions. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, OSDI '20. USENIX, November 2020.

[42] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, page 25–33, San Jose, California, 2006.

[43] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^{3}S: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, page 423–437. USENIX Association, 2008.

[44] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS checker: Combating bugs in distributed systems. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '07. USENIX Association, Apr. 2007.

[45] C. Lou, P. Huang, and S. Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS '19, Bertinoro, Italy, May 2019.

[46] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 559–574. USENIX Association, Feb. 2020.

[47] C. Lou, Y. Jing, and P. Huang. A promise is not a promise—demystifying and checking silent semantic violations in large distributed systems. Technical report, Johns Hopkins University, July 2022. https://orderlab.io/paper/oathkeeper-tr.pdf, Accessed July 2022.

[48] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 295–310, Monterey, California, 2015.

[49] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 370–384, Huntsville, Ontario, Canada, 2019.

[50] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 361–377, Monterey, California, 2015.

[51] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, USITS '03, Seattle, WA, Mar. 2003.

[52] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi. IASO: A fail-slow detection and mitigation framework for distributed storage services. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 47–62. USENIX Association, July 2019.

[53] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, EECS Department, University of California, Berkeley, Mar 2002.

[54] M. Rigger and Z. Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 667–682. USENIX Association, Nov. 2020.

[55] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Softw. Engg.*, 19(6):1665–1705, Dec. 2014.

[56] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early detection of configuration errors to reduce failure damage. In *Proceedings of the The 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, November 2016.

[57] N. Yaseen, B. Arzani, R. Beckett, S. Ciraci, and V. Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 701–718. USENIX Association, Nov. 2020.

[58] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 26–36, Szeged, Hungary, 2011.

RESIN: A Holistic Service for Dealing with Memory Leaks in Production Cloud Infrastructure

Chang Lou[1], Cong Chen[2], Peng Huang[1], Yingnong Dang[2], Si Qin[3], Xinsheng Yang[4], Xukun Li[2], Qingwei Lin[3], Murali Chintalapati[2]

[1]Johns Hopkins University [2]Microsoft Azure [3]Microsoft Research [4]Meta

Abstract

Memory leak is a notorious issue. Despite the extensive efforts, addressing memory leaks in large production cloud systems remains challenging. Existing solutions incur high overhead and/or suffer from high inaccuracies.

This paper presents RESIN, a solution designed to holistically address memory leaks in production cloud infrastructure. RESIN takes a divide-and-conquer approach to tackle the challenges. It performs a low-overhead detection first with a robust bucketization-based pivot scheme to identify suspicious leaking entities. It then takes live heap snapshots at appropriate time points in carefully sampled leak entities. RESIN analyzes the collected snapshots for leak diagnosis. Finally, RESIN automatically mitigates detected leaks.

RESIN has been running in production in Microsoft Azure for 3 years. It reports on average 24 leak tickets each month with high accuracy and low overhead, and provides effective diagnosis reports. Its results translate into a 41× reduction of VM reboots caused by low memory.

1 Introduction

Memory leak is a prevalent issue in software, from applications [13] to OS kernels and device drivers [46]. At Microsoft Azure, its infrastructure contains many complex software components running on a massive number of machines with various workloads. Unsurprisingly, these components encounter memory leak issues from time to time. When a process leaks memory, the direct consequence is performance degradation and crash. Worse still, its impact often affects other components running on the same machine, such as causing excessive paging, innocent processes being killed, and node reboots.

Memory leak is notoriously difficult to deal with, especially in a production cloud infrastructure setting. The issues are usually only triggered by rare conditions and occur slowly, thus they easily escape testing and failure detectors [20]. After leak symptoms are detected, it is time-consuming and sometimes impossible to reproduce them offline. Unlike other failures like crashes that have clear points to start diagnosis, developers are often clueless in finding the leak's root cause.

Extensive solutions have been proposed to detect memory leak bugs. One approach uses static analysis techniques [10, 15, 18, 36, 47] to analyze the software source code and deduce potential leaks. The second approach detects memory leaks dynamically by instrumenting a program and tracking the object references at runtime [16, 21, 25, 39, 49].

While helpful, these solutions are insufficient to address the memory leak challenges in Azure. Static approach is limited by the well-known accuracy and scalability issues with static analyses. It also only focuses on leaks in which allocated objects are unreachable [24]. If memory objects are reachable but never accessed again, it still incurs the consequences of leaks. Such leaks are hard to detect statically. Moreover, memory leaks in cloud infrastructure can be caused by cross-component contract violations, which require too much domain knowledge to recognize statically.

Dynamic approaches better fit Azure's requirements. However, while the existing dynamic detection solutions are generally more accurate, they are intrusive and require extensive instrumentations that are cumbersome to apply to complex components [16, 21]. They also incur high runtime overhead that is prohibitive for deployment in production [6].

In this paper, we present RESIN, an end-to-end service designed to holistically address memory leaks in large cloud infrastructure, from detection to diagnosis and mitigation. RESIN is highly scalable—it analyzes all the host software components, including kernels, drivers, and system processes, on millions of nodes in Azure. RESIN has low overhead while running in production environment. At the same time, RESIN provides good accuracy and helps developers pinpoint the root causes of memory leak issues.

Two key insights motivate the design of RESIN and enable it to achieve the above properties. First, the conundrum of existing solutions is in part because they mix detecting a leak and pinpointing the leak bug in one step, so they have to make trade-offs among accuracy, scalability, and overhead. In our experience, we should decompose the detection and pinpointing into multi-level stages to catch memory leaks at production scale. Second, taking a centralized service approach that

leverages low-level system mechanism is essential to support many components transparently in a non-intrusive way. It also enables gathering valuable information from many nodes in the cloud to address the accuracy challenges.

Based on these insights, RESIN performs non-intrusive, low-overhead leak detection first. When a process is suspected of experiencing leaks, RESIN triggers a live heap-snapshot mechanism to capture sufficient evidence and runs diagnosis. RESIN leverages kernel-level monitors and profilers as its building blocks, so it directly supports all the running processes without cumbersome integration. Furthermore, RESIN builds a centralized service that analyzes processes across all hosts in Azure fleet together to capture complex leaks.

A key challenge for dynamic leak detection is the highly noisy nature of memory usage in modern software affected by the workload characteristics. Using simple static thresholds can easily generate many false alarms or false negatives. For instance, in an impactful real-world cloud service outage caused by memory leaks, no alarm was triggered despite the existence of a memory monitoring service [4].

RESIN addresses this challenge by designing a robust *bucketization-based pivot* scheme. It aggregates the memory usages of processes across machines, and groups them into different buckets. Then by performing a pivot analysis on the process name, bucket, and other attributes, RESIN can reliably detect leaks without being prone to fragile thresholds. Essentially, we focus on analyzing a component's global memory usage behavior, rather than the microscope of an individual process. The rationale is that a true memory leak comes down to some buggy release. Although the memory usage of an individual process is highly dependent on workloads, the workload effect is likely canceled out when inspecting the usage of the same component running in all machines.

Once a suspicious memory leak is detected, RESIN activates the second stage of taking *live* heap snapshots of the suspected processes, which contain information about the active allocations and their stack traces. This stage is more heavyweight but provides more evidence to help developers confirm and diagnose the issue. Since a leak is often sporadic, RESIN aims to "hit" the leak again and capture useful evidence. It carefully chooses the snapshot time points so that the obtained snapshots have a high chance of localizing the root causes while minimizing the snapshot cost. Besides taking heap snapshots of the suspected leaking process, RESIN performs a *fingerprinting step* that periodically takes heap snapshots of representative processes to build a reference database. This reference database is used in the diagnosis algorithm to further improve the diagnosis accuracy.

Finally, RESIN automatically mitigates a detected leak to minimize its impact on the service availability and performance. The mitigation engine in RESIN leverages the information from the detection and diagnosis engines, and determines the appropriate actions to resolve the leak symptoms while developers investigate the root causes and fixes.

RESIN has been running in production in Azure for more than 3 years. RESIN reported many memory leaks, helped developers diagnose the issues, and automatically mitigated the leaks before their impact becomes visible to customers. Within the recent year at the timing of writing, the unexpected VM reboots in Azure caused by out of memory are reduced by 41×, and the new VM allocation errors due to low memory are reduced by 10×. In addition, no severe outages in 2020 and 2021 at Azure were caused by memory leaks.

In summary, the main contributions of this work are:

- A holistic memory leak solution for cloud infrastructure.
- A novel bucketization-based pivot scheme to robustly detect memory leaks with low overhead.
- A live heap snapshot algorithm to effectively capture evidence in production and diagnose memory leaks.
- A lightweight automated leak mitigation design.
- Deployment of RESIN in a production cloud service.

2 Background and Motivation

2.1 Host Memory Compositions

In IaaS cloud infrastructure, servers are equipped with large memory, a significant portion of which is used by the virtual machines (VMs), while the other portion is used by the host software. The latter includes the hypervisor, host OS kernel, drivers, system processes, and various host agents, *e.g.*, an agent that manages networking of the VMs. In this work, we focus on memory leak issues in host software, not leaks in customer VMs. Unless otherwise specified, the kernel, drivers and processes hereafter refer to those in host software stack.

Leaks in the host software can cause severe performance degradation and even instability of the host OS. They can further impact the running VMs, because memory between VMs and the host is not strictly partitioned, typically controlled by a soft threshold [45]. They can also cause potential VM start-up failures due to insufficient physical memory available.

The host memory is divided into user-mode memory and kernel memory. The host OS in Azure's infrastructure distinguishes four states for pages in a process' virtual memory: *free*, *reserved* (for future use but no physical page is allocated), *committed* (memory has been allocated from physical memory or paging files), and *shared*. For memory leak detection, we only need to consider pages in the committed and shared states. For kernel memory, the kernel creates two types of memory pools: *non-paged* pools and *paged* pools. Virtual memory in the non-paged pool is guaranteed to reside in physical memory as long as the kernel objects are alive, whereas memory in paged pool can be paged out. Memory leaks in the kernel can happen in both types of pools.

2.2 Memory Leaks

Memory leak occurs when heap-allocated objects are not freed at appropriate time. It is manifested in two forms: (i) *unreachable* leak, in which an allocated object is no longer

```
1  // ConfigMonitorThread          5-sec timeout, previously
2  while (cm->running) {           it was set to INFINITE
3    waitStatus = WaitForSingleObject(
4                      fileChangeHandle, 5 * 1000);
5    if (waitStatus == WAIT_OBJECT_0) {
6      // object is signaled, config file has changed
7      ::Sleep(200);
8      cm->ReadConfig(); // read the file
9  +   if (!FindNextChangeNotification(fileChangeHandle))
10 +     throw ServerBaseException(
11 +         "Failed to get handle to config directory");
12   }
13 - FindNextChangeNotification(fileChangeHandle);
14 }
```

Figure 1: A production memory leak example in Azure from a host process that caused leaks of objects allocated at the kernel side.

reachable from the root objects such as global and stack variables; (ii) *forgotten* leak, in which an allocated object is still reachable but no longer accessed. The first type does not occur in managed languages like Java. For the second type, since the program still keeps references to the leaked object, it cannot be reclaimed even with managed languages [12]. Such leak is challenging to be detected because whether an object will be accessed in the remaining execution is undecidable. Thus, a leak detection solution can only output conservative (correct) answers, *e.g.*, the objects that are definitely dead at a given time point, or approximate answers (which may be incorrect) such as inferring based on the object's *staleness* [17].

Memory leaks in cloud software have further complications. For instance, while existing solutions focus on detecting leaks in an individual component, a memory leak in cloud infrastructure often happens because of API contract violations between different components, which is not well addressed. This type of leak is hard to expose in pre-production environment, because software components are often tested separately and integration testing cannot cover all possible interactions. Slow leaks also unlikely get detected due to testing time constraints.

Figure 1 shows a real example of such a leak in Azure (this case was successfully caught by RESIN). The process has a thread that monitors the configuration file updates using `WaitForSingleObject` with a 5-second timeout. In each loop iteration, it calls the `FindNextChangeNotification` API (line 13). Each invocation causes the kernel to allocate I/O request kernel objects from the non-paged pool memory. The contract of the `FindNextChangeNotification` is that it must be followed by a call to a wait function, and if the wait function returns for any reason other than the change notification handle being signaled (*e.g.*, timeout), the wait must be retried. In this case, although the process calls the wait function, it unconditionally calls `FindNextChangeNotification` even if the wait returns timeout. Thus, the kernel objects are allocated every 5 seconds without being cleaned up. In this incident, the culprit process' memory usage was not high. The kernel was experiencing memory leak in its non-paged pool, not because of kernel bugs but rather the improper API usage in the process' code. This memory leak was introduced during a bug fix for another issue: previously the process waits for the updates using an `INFINITE` parameter in line 4, but this caused service restart operations to be blocked, so developers changed the wait parameter to a timeout of 5 seconds.

2.3 Requirements

There are several challenges and requirements for addressing memory leaks in cloud infrastructure software:

- *Highly scalable.* Cloud system is large in the number of components, codebase size, and deployment scale.
- *Versatile.* Memory leaks in cloud infrastructure manifest themselves in various ways—in processes, kernel, unreachable leaks, forgotten leaks, cross-component leaks, *etc.*
- *Non-intrusive and low-overhead.* Solutions that require intrusive modifications or incur high runtime overhead are hard to be deployed in production.
- *Accurate.* True leaks should be detected. False positives should be minimized, because they would cause developers to waste significant time investigating false issues.
- *Timely.* If the leak detection is too slow, significant damage to customers may already occur.
- *End-To-End.* Only alerting memory leaks is insufficient. Developers also need considerable help in confirming the issue, pinpointing the root cause, and mitigating the leak.

Additional constraints include generality and efforts of integrating a solution. The software components in cloud infrastructure are written in different programming paradigms, and may depend on proprietary libraries. The millions of nodes in Azure also have heterogeneity with different OSes, libraries, and hardware versions. Supporting all of these varieties is challenging. For example, we made an experimental effort of integrating the LeakSanitizer [1], a popular run-time memory leak detector from the LLVM project, into one Azure host component's codebase. The integration effort was difficult (took one person month) due to complex compilation flags, and library compatibility issues. The MSVC compiler's full support for LeakSanitizer is still pending [3].

3 Overview of RESIN

Despite the extensive efforts to address memory leaks in conventional settings, they are insufficient to satisfy the unique requirements for tackling memory leaks in large cloud infrastructure (Section 2.3). To address this gap, we propose RESIN. RESIN is a holistic system running in the Azure production infrastructure to detect memory leaks in host software and provide diagnosis support to developers easily pinpointing the leak's root causes. RESIN further performs automatic leak mitigation to reduce the impact of detected leaks.

Approach A large cloud infrastructure can have hundreds of components owned by different teams. Prior to RESIN, tackling memory leaks in Azure is a team-by-team effort. Some teams started investigating after incident reports about slow or failing VMs, and developers discovered leak bugs in their components during manual investigation. Some teams added telemetry monitors in their testing cluster and used

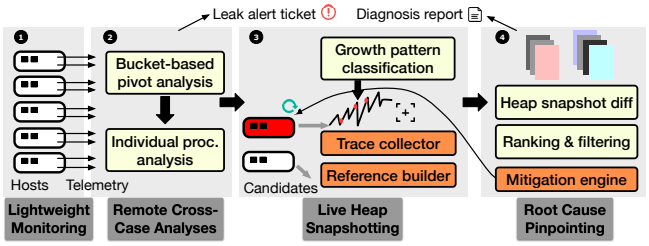


Figure 2: Workflow of the RESIN system.

hard-coded thresholds to trigger leak alerts in testing. Similarly, diagnosing leaks in Azure used to rely on developers to manually inspect the leaking nodes and run profiling tools. These individual practices were tedious and costed repeated engineering efforts. They also incurred significant false positives and could not handle cross-component leak issues.

RESIN takes a centralized approach instead. It does not require access to a component's source code, nor extensive instrumentation or re-compilation. RESIN uses a monitoring agent to each host that leverages low-level OS features to collect memory telemetry data. It automatically supports all components including the kernel. The data analysis is offloaded to a remote service, which minimizes the overhead to the hosts. By aggregating data from different hosts, RESIN can run more sophisticated analyses to catch complex leaks.

In addition, RESIN decomposes and tackles the memory leak problem in multi-level stages. It performs lightweight leak detection first and triggers more in-depth inspections on the fly when necessary for confirmation and diagnosis. This divide-and-conquer approach allows RESIN to achieve low overhead, high accuracy, and scalability together.

Workflow Figure 2 shows the workflow of RESIN. It starts with low-overhead monitoring (❶) at each host. A remote service analyzes (❷) the collected data across different hosts using a bucketization-pivot scheme. If a bucket is suspected of leaking, RESIN triggers an analysis on the process instances from that bucket. After the two steps identify a highly suspicious software component, RESIN automatically generates an alert ticket for that component along with a list of leaking process instances belonging to that component. Meanwhile, RESIN performs live heap snapshotting (❸) for the suspected processes. RESIN carefully chooses the snapshotting time using a growth pattern based algorithm to ensure the collected snapshots would be helpful. RESIN also samples normal processes to take regular heap snapshots and build a reference database. After generating multiple heap snapshots, RESIN tries to pinpoint the root causes (❹) by running a diagnosis algorithm on the snapshots. The analysis report will be attached to the alert ticket thread to assist developers. Finally, RESIN automatically mitigates the leaking processes.

4 Design of Leak Detection

In this section, we describe the RESIN's design for detecting memory leaks. In existing literature, the term "detection" refers to detect both (i) if a program or a process has a leak, and (ii) the bug in code or leaked objects. RESIN separates these as two tasks and uses the term detection to refer to (i) specifically. The diagnosis component (Section 5) targets (ii).

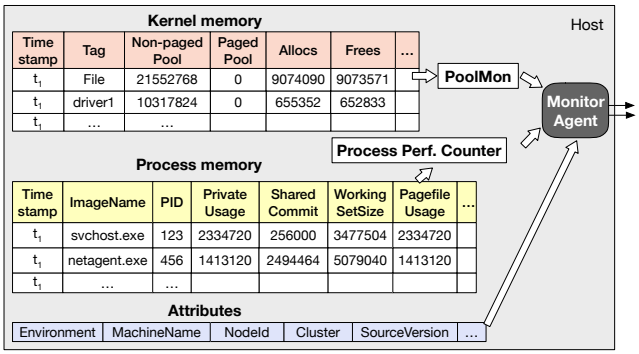


Figure 3: Monitor agent in each node collecting memory usage data.

4.1 Challenges

RESIN needs to address several challenges. First, cloud infrastructure software has highly noisy memory usage due to changing workloads and interference in the environment. Using static thresholds would generate many false positives. Standard anomaly detection algorithms [40, 41] do not work well either, because it is common for a component to exhibit memory usage spikes that are not leaks but legitimate increases in handling certain workloads.

Second, memory leaks in production systems are usually fail-slow faults [14] that last days, weeks, to even months (rapid leaks are likely caught in testing or deployment). Inspecting memory usage in a short time window would miss these slow leaks. It is necessary but challenging to capture gradual changes over a long period and still raise timely alerts.

Third, given the scale of Azure, collecting fine-grained data for a long time is impractical because of storage and overhead concerns. Therefore, RESIN can only collect limited, coarse-grained data and must work well under this constraint. Still, even with coarse-grained signals, the data volume is enormous. The detection algorithms must run efficiently.

4.2 Lightweight Memory Usage Monitoring

RESIN deploys a privileged monitoring agent on each host (Figure 3). This agent communicates with the host OS to track memory usage. It collects both kernel memory usage and per-process memory usage. The kernel usage is obtained from a pool monitor kernel module (`PoolMon`), and includes the usages of non-paged memory pool and paged memory pool for each *tag*. The tag is passed as an argument by the callers of the kernel allocation API [32] and represents a sub-system that has requested memory from the kernel allocator, *e.g.*, the file system, a driver. The per-process usage is obtained by querying the per-process performance counters from the host. It includes breakdowns of a process's memory, such as the private commit, working set size, paging file usage, *etc.*

We collect the memory usage breakdowns and tags instead of simply a single total memory usage metric, because mixing

different memory usage sources can introduce noises and miss important changes. For example, a 20 MB increase can be a leak for a driver but may be negligible for another component. Reporting the specific memory portion or tag that is leaking helps developers localize the buggy code. The breakdowns also help RESIN take more effective mitigation actions.

In addition to memory usages, the monitoring agent also records attributes such as the software version, hardware generations, node id, and cluster id. The attributes are used during leak detection analyses to increase accuracy. RESIN includes the common attributes of the leaked process in the detection report to give developers troubleshooting hints.

The monitoring agent is scheduled to run every 5 minutes. However, the data points from different hosts may not be perfectly synchronized. Some special events in a host such as node reboots also introduce missing or invalid data. Therefore, RESIN aggregates the time-series data into hourly granularity by removing extreme outliers and computing the mean of the remaining data points. This pre-processing step reduces the noises as well as the data volume. Using an hourly window is not too coarse-grained because most software components in cloud infrastructure are long running, and production leaks typically occur in a large time scale.

4.3 Detection Algorithms

RESIN uses a two-level scheme to detect memory leak symptoms: a global bucketing-based pivot analysis to identify suspicious components, and a local individual process leak detection to identify leaking processes. The detection output includes the suspected component, the list of top leaking processes of that component, the leak start and end times, severity scores, *etc.* The detection algorithms are language agnostic.

4.3.1 Bucketization-based Pivot Analysis

To address the challenges described in Section 4.1, our insight is that we should inspect at the *component* granularity across processes. This is because although an individual process' memory usage is influenced by workloads and highly noisy, the noises can be "canceled out" *en masse*. For a normal component, its process instances on different hosts may experience different workload effect at any time slice. But for a leaky component, the memory leak must be caused by some buggy release. Therefore, its processes should exhibit some global trend at certain time slices despite the workload effect.

Based on this insight, we design a simple yet robust bucketization-based pivot detection scheme (Figure 4). RESIN first groups the raw memory usage telemetry data into a number of buckets. In our implementation, we use 20 buckets (50 MB, 100 MB, 200 MB, ..., 40 GB, 50 GB). RESIN then applies pivoting to the data with a unique attribute tuple as the index and memory usage bucket as columns. The attribute tuple is (`ProcessImageName`, `ServiceName`) for user-level software, and (`TagName`, `PoolType`) for kernel subsystems, where `Type` is paged memory or non-paged memory. The aggregation function is the count of distinct nodes. Thus, each summary

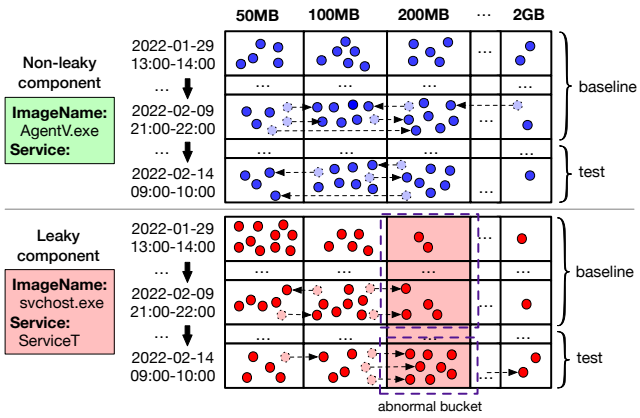


Figure 4: Group the memory usage into buckets and pivot by image name, service, and bucket size. Each circle represents one process. Shaded circle represents a process moving to another bucket.

cell represents the number of nodes that have running processes with a particular attribute tuple and these processes' memory usages fall into the specific bucket. RESIN computes the summary periodically and incrementally for data in each time interval. The results are saved into a database table.

We basically transform the memory usage data into summary about numbers of nodes in different buckets, which can more robustly represent the trends and tolerate noises due to workload effect (*e.g.*, the non-leaky component in Figure 4). RESIN then runs anomaly detection on the time-series data of each bucket for each component. It uses the most recent time period of summary data (default 15 days), with the first $2/3$ portion as the baseline and the remaining data points as the test. If a bucket's test period has data points that exceed the $\mu + 3\sigma$ of the baseline data (μ and σ represent the mean and standard deviation of the distribution), it is considered to be an anomaly. The start time and end time in the test period when the node count becomes the outlier are recorded.

One caveat is that if many processes of a component experience a sudden *drop* in memory usage, the node count would shift from a higher bucket to a lower bucket. But the lower bucket's node count significant increase is not an anomaly. To handle such scenarios, RESIN calculates cumulative bucket values during the anomaly detection. In other words, if the 100 MB bucket has a node count of n, it means there are n nodes that have the particular processes with memory usage equal to or larger than 100 MB, including processes (if any) that fall into the 200 MB bucket. In this way, a significant increase in a bucket almost always suggests an anomaly.

The bucketization approach also helps address the computation challenges. Before introducing this approach, it can take RESIN more than one day to run anomaly detection on the enormous data points from millions of nodes. After the pivot summary, RESIN only needs to run anomaly detection for the time-series data in each bucket, which can finish in less than one hour for all data (even without parallelization).

RESIN calculates a severity score for each bucket based on the deviations and node count in the bucket. It considers a

component is leaking based on a <*size_mb*, *score*> threshold: if a bucket is of size equal to or larger than *size_mb* and its severity score exceeds *score*. RESIN generates intermediate reports for the abnormal buckets. It de-duplicates the intermediate reports by only keeping the one for the largest bucket of a unique attribute tuple, and generates a ticket for that report.

4.3.2 Localizing Individual Processes

The bucketization pivot analysis works at the component granularity. RESIN uses a second-level detection scheme that works at the process[1] granularity. The motivation for this scheme is that a component has many process instances. It is important to localize the truly leaking processes in the alerting bucket. If we simply include all the processes in the abnormal bucket, developers can waste significant effort investigating innocent processes that fall into that bucket by coincidence.

The second-level detection scheme computes the leak likelihood and severity for a process based on its memory usage. RESIN uses all the memory usage data of a component in the most recent month to train two parametric models: (i) the *absolute usage model* and (ii) the *usage difference model*. Since different clusters and regions can exhibit drastically different characteristics, the tool builds separate models for each combination of region name and cluster type for a component.

Let $U_c(n_i,t_j)$ denote the memory usage value for a process of component c on node n_i at time t_j. RESIN assumes *absolute usage* $U_c(n_i,t_j)$ follows a Gaussian distribution $\mathcal{N}(\mu_1,\sigma_1^2)$ and fits the memory usage data by calculating the maximum likelihood estimators for μ_1 and σ_1. The absolute memory usage values can be severely distorted by occasional events such as VM creations. To account for such events, we consider the differential memory usage, *i.e.*, $\Delta U_c(n_i,t_j) = U_c(n_i,t_j) - U_c(n_i,t_{j-1})$. Based on our observations, when noisy events such as VM creations occur, $\Delta U_c(n_i,t_j)$ usually significantly deviates from its normal range. Thus, RESIN also builds a parametric Gaussian distribution $\mathcal{N}(\mu_2,\sigma_2^2)$ model for *usage difference* $\Delta U_c(n_i,t_j)$ and calculates the μ_2 and σ_2.

With the offline models, RESIN uses a *moving suspicious interval* algorithm (Algorithm 1) to examine a suspected process' memory usage in *real time*. This algorithm works by keeping a *suspicious leak time interval* $[T_0,T_1]$. The basic idea is to assume the leak still continues at the end of the time series and try to find the earliest time the leak trend starts by skipping over low-confidence points. This interval is initialized as $[t_1,t_1]$ upon reading the first data point in a time series. At the j-th step, RESIN reads $U_c(n_i,t_j)$, calculates the $\Delta U_c(n_i,t_j)$, and adjusts the time interval by moving T_1 and update T_0 adaptively. If $\Delta U_c(n_i,t_j)$ has a significant increase or drop (based on the 3-sigma rule for μ_2 and σ_2), T_0 is updated to t_j because the system status is likely changed by some event. If $U_c(n_i,t_j)$ is lower than $U_c(n_i,T_0)$ or there are few increasing points in the current interval, T_0 is also updated to t_j because a leaking trend should have enough increasing values. For other situations, we keep the T_0 intact. The loop stops when T_1 hits the last time point t_n. If the final T_0 is equal to T_1, this process is not considered as leaking.

[1] Here we use the term "process" to also include a running instance of a kernel subsystem in a particular host for kernel memory leak detection.

Algorithm 1: Moving suspicious interval algorithm

Input: $U_c(n_i,t)$: time-series memory usage for node n_i of component c; $\mathcal{N}(\mu_2,\sigma_2^2)$: offline usage difference model for component c.
Output: T_0, T_1: leak start and end time; no leak if T_0==T_1. N_{inc}: number of increasing data points

```
t_n ← max(t) in U_c(n_i,t), N_inc ← 0
T_0 ← t_1, T_1 ← t_1
for j ← 2 to n do
    T_1 ← t_j
    ΔU_c(n_i,t_j) ← U_c(n_i,t_j) - U_c(n_i,t_{j-1})
    if IsOutlier(ΔU_c,N) || U_c(n_i,t_j) < U_c(n_i,T_0) || N_inc/n < ε then
        T_0 ← t_j
    if T_0 == T_1 then
        N_inc ← 0          /* empty interval, no leak, reset */
    else if IsLarger(U_c(n_i,t_j),U_c(n_i,t_{j-1}),N) then
        N_inc ← N_inc + 1  /* a new increasing data point */
return T_0, T_1, N_inc
```

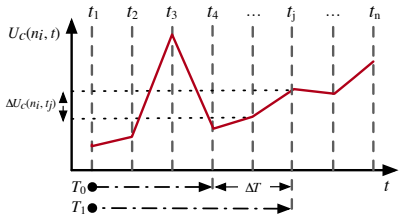


Figure 5: Applying the moving suspicious interval algorithm.

Figure 5 shows an example of applying the algorithm. $[T_0,T_1]$ are initially set to $[t_1,t_1]$. When T_1 is set to t_2, $\Delta U_c(n_i,t_j)$ is positive thus we keep T_0 unchanged and continue to move T_1 forward. When T_1 is set to t_3, $\Delta U_c(n_i,t_j)$ is an outlier in the offline model $(\mathcal{N},\mu_2,\sigma_2^2)$, which we consider an occasional event instead of a leak. We reset T_0 to t_3 accordingly. When T_1 is set to t_4, $U_c(n_i,t_j)$ is significantly lower than $U_c(n_i,T_0)$ thus we also reset T_0 to t_4. After t_4 we did not encounter scenarios to reset T_0 (the memory usage drops slightly later but it is unnecessary to reset for such cases), so eventually T_1 reaches the end t_n, and $[T_0,T_1]$ is $[t_4,t_n]$.

RESIN calculates a severity score (Equation 1) for a process to indicate its leak probability and impact. Several factors are considered, including the normalized memory usage difference ($\Delta U_c = U_c(n_i,t_n) - U_c(n_i,t_1)$), the length of the suspicious leak interval ($\Delta T = T_1 - T_0$ in the unit of month), the increasing rate (number of increasing data points over the number of all data points), and the final memory usage at t_n.

$$SevScore = \frac{\Delta U_c}{\sigma_1} + \frac{N_{inc}}{n} + \Delta T + \frac{1}{1+e^{-U_c(n_i,t_n)/\mu_1}} \qquad (1)$$

For efficiency, the above analyses are run proactively. When the bucketization-based pivot detection step identifies a leaking bucket, RESIN triggers the individual process analysis for the processes in the bucket and can usually inspect the results without waiting. It outputs the suspected leaking processes, the leak start and end time, and the severity scores.

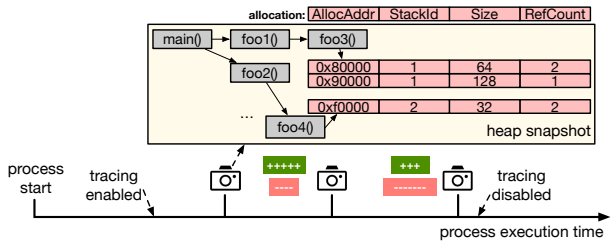


Figure 6: Periodic heap snapshot collection.

5 Diagnosis of Detected Leaks

Only detecting a leak is not enough. Without sufficient evidence and diagnosis support, developers are likely stuck in confirming and diagnosing the issue. RESIN designs a solution that automatically takes *live* heap snapshots and analyzes the snapshots to pinpoint the root cause of a detected leak.

5.1 Background: Heap Snapshot

RESIN provides diagnosis information at stack trace granularity. In our experience, if developers are presented with a stack trace containing the problematic allocations, they can often quickly debug the issue. RESIN leverages the Windows heap manager's snapshot capability to perform live profiling. The heap manager exposes APIs such as `HeapAlloc`, `HeapReAlloc`, and `HeapFree`, which are used by applications and C/C++ runtime to allocate heap objects. Thus the heap manager has the ability to collect the heap allocation sizes and stack traces.

RESIN uses the Windows Performance Recorder [2] to notify the kernel to start tracing heap allocations, typically for a specific process ID and occasionally for an image name (which enables tracing for all processes with the image). Then RESIN instructs the heap manager to take a snapshot at a certain time. Our current heap snapshotting mainly focuses on C/C++, which are the primary language choices for host software on Azure. Extending to other languages would take extra effort but are still straightforward, as their runtime typically already provides the functionality to capture allocation events.

To minimize overhead, the heap manager only stores limited information in each snapshot. Specifically, it stores (1) the stack trace and size for each *active* allocation after the tracing was enabled (if an allocation has been freed, no information is stored), (2) the total allocation sizes for each unique stack trace, and (3) the number of times a unique stack trace is invoked. It does *not* store more detailed information such as the allocation time or a pointer graph.

The information in a single snapshot is usually too noisy, as it includes all active allocations from the tracing start to the snapshot point. To get more accurate information for a time window, RESIN periodically takes *multiple* heap snapshots (Figure 6) to increase the chance of capturing truly leaking allocations between snapshots. RESIN uploads the snapshot files to a remote storage service. The diagnosis engine uses these snapshots to deduce the leaking allocation points.

5.2 Choosing Candidate Hosts to Profile

Picking the right hosts to take heap snapshot is vital for diagnosis effectiveness. Because heap snapshot incurs overhead, RESIN cannot afford to enable snapshot on all hosts containing the leaking processes the detection engine outputs. Simply choosing the hosts randomly is not a good strategy either, because the workloads on different hosts vary widely. For the same leak bug, it can exhibit in quite different patterns on different hosts. Thus, we may choose a candidate host in which the buggy allocations are triggered rarely.

We rank the candidate hosts in the suspected list based on three factors: 1) *severity*: choose processes with higher severity scores as described in Section 4.3.2, since more obvious symptoms suggest a better chance to be diagnosed; 2) *noisiness*: choose processes with a clearer growth pattern, which we will discuss in more detail in Section 5.3; 3) *impact*: choose hosts that have fewer user activities to minimize impact of profiling events. By default RESIN triggers snapshot collection for the top three hosts in the list in case the collection fails unexpectedly (*e.g.*, due to target process restart).

5.3 Deciding Trace Collection Strategy

With the candidate hosts selected, the next step is to decide if a *new* leak happens in the most recent snapshot interval and whether to take the snapshots. This step is different from the analysis in Section 4.3, which only finds leaking processes in *past* time. The decision making has two main challenges.

First, many production leaks are only triggered by specific events. Some leaks only occur once in several days. If we take snapshots at other times, the collected traces would not be helpful. To ensure rare leaks are captured, RESIN attaches the profiling workflow to the process for a long time and periodically (every half hour) takes snapshots in hope of capturing the leak. However, we cannot afford to keep uploading snapshots due to storage and overhead concerns. RESIN addresses the challenge with a *long-term, trigger-based* strategy: it uses a circular buffer that only keeps the most recently taken snapshots, and completes tracing once certain trigger is met.

Second, how to decide when the trace collection should complete, *i.e.*, the trigger. At the completion time, we should ideally (1) have snapshot(s) containing the buggy allocation; (2) have snapshot(s) for non-leaking scenarios; (3) minimize noisy allocations in the snapshot(s). One potential trigger is to complete the collection once the memory usage difference exceeds some threshold. This trigger can easily complete the tracing prematurely (fails to capture the buggy allocation) due to a legitimate memory usage spike, and/or produces snapshots that have many noisy allocations and mislead diagnosis.

To gain some insights on how to choose the triggers, we study the memory usage data of 51 real leak cases. Interestingly, most cases fall into three common patterns (and a mixture of them). Additionally, one leaking process in a specific host often has a *consistent* pattern. In 63% of the cases, the leaking process shows a *steady* pattern. One example is

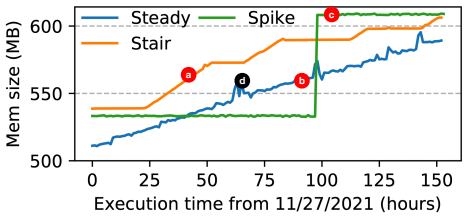


Figure 7: Memory growth patterns and completion point choices. Each data point is real memory usage from production processes.

Pattern	Characteristics	Completion Trigger
Steady	Almost linear growth	$R_j^2 > \lambda_a$.
Stair	Steady growing and flat curves alternately appears.	$\lvert 1 - slope_j/slope_k \rvert < \lambda_b$ && $R_j^2 < \lambda_c$
Spike	A few large allocations in a short period of time	$\Delta U_c(n_i,t_j)/\Delta U_c(n_i,t_k) > \lambda_d$

Table 1: Leak patterns, characteristics and their completion triggers.

the bug shown in Figure 1, in which the leak exists in periodical update tasks. The other two common patterns are *stair* (the memory usage occasionally grows only when the procedure containing leaks is activated) and *spike* (the leak only occurs once in a while due to rare events). Each pattern has its unique usage growth characteristics and clear completion point candidates ⓐ, ⓑ, ⓒ shown in Figure 7. ⓓ is not a good completion point as it is right after a period of some noises.

Guided by the findings, RESIN takes a pattern-based approach to decide the trace completion triggers. It uses the target process' memory usage data in the most recent week and classifies it into one of the three patterns. Specifically, RESIN first identifies nearly flat segments in the time-series data and removes them. It then performs linear regression on the remaining growing segments and outputs results including the slope $slope_k$, coefficient of determination R^2, and absolute memory usage increase $\Delta U_c(n_i,t_k)$. If the data contains no flat segments, RESIN marks it as a *steady* pattern. If the data has flat segments in between growing segments, it is marked as a *stair* pattern. If a large increase in memory usage only occurs in a few data points, it is marked as a *spike* pattern.

After the pattern is classified, the workflow starts to monitor and analyze the recent memory usage. We compute the $slope_j$, R_j^2, and $\Delta U_c(n_i,t_j)$ for the most recent six hours and check the pattern's completion trigger based on rules listed in Table 1 (λ_a, λ_b, λ_c and λ_d are set to 0.8, 0.1, 0.1, and 0.5, respectively). Once the trigger is satisfied, RESIN stops tracing and uploads the trace file that contains the most recent few snapshots. It ensures each trace has at least three snapshots.

5.4 Collecting Reference Snapshots

One challenge in using snapshots for diagnosis is the presence of many noisy but benign allocations. Even with multiple snapshots, they may remain active and mislead the diagnosis. RESIN collects *reference snapshots* to address this challenge.

For the reference snapshots to be useful, they should be comparable to the snapshots from the leaking process. A poor choice of a reference snapshot may be even counter-

Algorithm 2: Heap snapshot diagnosis algorithm

Input: A_{n-1},A_n: sets of allocations in two heap snapshots, S^r: a list of outstanding stacks from reference hosts, *pattern*: classified pattern, *estimate_leak*: upper bound of estimated leaking size
Output: S^o: a list of top N stack traces that likely caused leaks

```
S^o ← [], S^diff ← []
S^n ← A_n.groupBy(alloc=>alloc.stackid)
S^{n-1} ← A_{n-1}.groupBy(alloc=>alloc.stackid)
foreach stack ∈ S^n do
    if stack ∈ S^{n-1} then A^diff ← S^n[stack.id] \ S^{n-1}[stack.id]
    else A^diff ← S^n[stack.id]
    if A^diff ≠ ∅ then
        foreach a ∈ A^diff do stack.size ← stack.size+a.size
        S^diff.add(stack)
S^diff.orderBy(stack=>stack.size)
if pattern ≠ SPIKE then
    S^diff.removeAll(stack=> stack.size > estimate_leak)
S^diff.removeAll(stack=> stack ∈ S^r)    /* filter references */
S^o ← S^diff.top(N)                      /* only keep top N stack traces */
return S^o
```

productive and filter out the culprit allocations. RESIN uses a periodical *fingerprinting* process to build reference snapshots. It randomly samples hosts for common leaking services to take heap snapshots. We currently define the fingerprints to be the attribute tuple `(cluster_id, OS version, service version, date)`. This is based on our observations on the locality of memory leaks. These snapshots are saved in a reference database and cleaned up when they become stale.

At the diagnosis stage, after RESIN chooses the candidate leaking hosts to profile (Section 5.2), RESIN checks if the database already has reference snapshots with similar fingerprints. If not, RESIN triggers reference collection. It first scans the qualified hosts (not in the detection engine's suspicious list and have similar fingerprints to the leaking hosts) and samples a few that have active memory activities and modest memory usage. Then RESIN applies the growth pattern analysis (Section 5.3) to check if this host is leaking. If not, it takes snapshots and uploads the traces to the reference database.

5.5 Trace Analyses for Diagnosis

The next step is to analyze the collected snapshots to output the root cause stack traces. The challenge is to handle many noisy allocations and localize the buggy allocations.

RESIN designs a diagnosis algorithm listed in Algorithm 2. The inputs are the allocations from the two most recent snapshots of a trace file (A_{n-1}, A_n), stack traces from reference snapshots, and the estimated leaked size upper bound calculated in the pattern analysis. For the *steady* and *stair* patterns, we estimate the leaked size upper bound by multiplying the slope with the time interval of the growing segments and a coefficient (by default 2). The goal is to find the stack traces that allocate objects of sizes closest to the estimated leak.

The diagnosis engine first groups allocations in A_{n-1}, A_n by the stack trace id and get two maps S^n and S^{n-1}. Each map value is all the allocations that come from a stack trace. It then traverses each stack trace in S^n to calculate the aggre-

gated allocation size. The engine then identifies stack traces that contain unique allocations, and ranks these traces based by their allocated object sizes. Stack traces allocating sizes larger than the estimated leak size are likely noises and thus removed. Finally, the diagnosis engine cross-checks the reference snapshots to filter out benign stack traces. If the output list is empty, the engine repeats the analysis for the next snapshot pairs (A_{n-2}, A_{n-1}), *etc.*

6 Mitigating Leaks

When a memory leak is detected, it can take time for developers to come up with and deploy the bug fix. To avoid further customer impact, RESIN attempts to automatically mitigate the detected leak issues. Depending on the nature of the memory leak, mitigation can be done in several ways. Rebooting the host OS in general can mitigate all kinds of leaks. However, this is costly and potentially causes VM downtime.

RESIN leverages the results from its detection engine and uses a rule-based decision tree to choose a mitigation action that can minimize the impact. If the memory leak is localized to a single process or Windows service, and this process or service is not required to be always alive to provide services to customers, RESIN attempts the lightest mitigation by simply restarting the process or Windows service.

For some processes, the mitigation requires additional steps. RESIN allows component teams to define custom scripts and invoking conditions. If the leak is located in buggy drivers, RESIN unloads and reloads the driver to mitigate the issue.

For safety, RESIN uses allowlists for each action category to make sure auto-mitigation is not misused. It defines an initial allowlist for the processes and drivers that are known to be safe to restart. A feature team can opt in auto-mitigation by adding the name of the process or tag to the allowlist.

For leaks in the OS kernel memory such as I/O request objects and file objects, if the detection engine can attribute the leak to a process or a service, RESIN attempts to restart the culprit process or service. This action is usually effective because it allows the leaked kernel objects to be properly freed without the need to reboot the OS.

OS reboot will resolve any software memory leak but takes a much longer time and can cause VM downtime. Thus, it is the last resort when a leak cannot be mitigated by the above actions or the name is not in the allowlist. RESIN checks if the host is empty and does the OS reboot if so. Empty hosts could also leak memory due to past user activities or current background processes. For a non-empty host, RESIN first performs live VM migrations [8]. Then it attempts a kernel soft reboot, which skips hardware initialization. If the soft reboot is ineffective, a full OS reboot is performed.

To minimize the impact of mitigation actions, RESIN closely monitors the leaking hosts. It prioritizes the actions on 1) nodes that fire low-memory related events, such as E2004 (low virtual memory) from the Windows resource exhaustion detector, E3122 (not enough memory to start VM) from Hyper-V; 2) nodes in regions with capacity issues; 3) nodes with host memory reservation overage; 4) nodes ordered by the leak size, leaking rate, and the predicted time-to-failure.

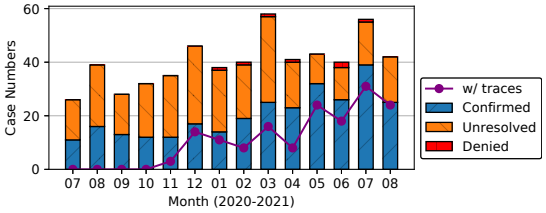


Figure 8: Memory leak cases RESIN detected and reported.

RESIN stops applying mitigation actions to a target when the detection engine no longer considers the target leaking. This typically occurs *without* manual intervention. For example, after developers identify the root cause and apply a fix, the leak symptom disappears, so RESIN stops the mitigation. Sometimes, after a mitigation action, the leak symptom no longer re-appears, which naturally stops further mitigation.

RESIN also coordinates with its diagnosis engine (Section 5) in performing the mitigation actions. If the diagnosis engine plans to or is taking heap snapshots for a candidate host, RESIN defers the mitigation actions to avoid losing the critical opportunities for capturing the leaking allocations.

7 Evaluation

Our evaluation answers several questions: (1) how effective is RESIN in detecting memory leaks? (2) how accurate is the detection? (3) can RESIN help developers diagnose and mitigate leaks? (4) what is the overhead of trace collection?

7.1 Deployment Status and Scale

RESIN has been running in production in Azure since late 2018. It covers millions of hosts, over 600 different host processes and over 800 different kernel pool tags daily. The detection engine in RESIN analyzes more than 10 TB memory usage data every day. The diagnosis module collects 56 trace files on average (10–200 MB) daily. Every month, the mitigation engine performs a median of 1,592 process restarts, 1,290 kernel soft reboots, and 4,649 node reboots.

7.2 Detecting Production Memory Leaks

Azure has various solutions that help eliminate memory leak bugs before production, including code reviews, static bug finding tools, testing, and safe deployment policies. As a result, only complex memory leak bugs occasionally escape these solutions. RESIN serves as the last defense to effectively catch these bugs in production.

Figure 8 shows the memory leak tickets RESIN reported in Azure from July 2020 to August 2021. Overall, RESIN reported 564 tickets in 14 months, among which developers explicitly resolved 291 tickets.

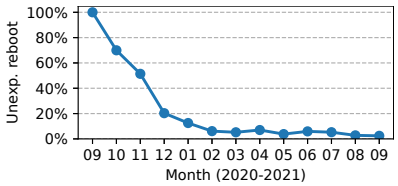


Figure 9: Unexpected VM reboot.

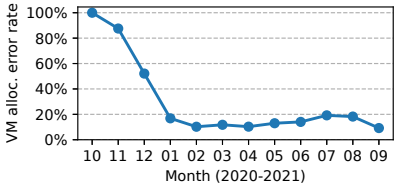


Figure 10: VM allocation error rate.

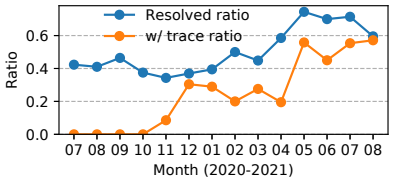


Figure 11: Ticket resolved and w/ trace ratio.

7.3 End-to-End Impact

The end-to-end benefits brought by RESIN are clearly demonstrated by two key metrics: (1) *VM unexpected reboot*[2]: the average number of reboots per one hundred thousand hosts per day due to low memory; (2) *VM allocation error*: the ratio of erroneous VM allocation requests due to low memory.

As shown in Figures 9 and 10 (data is normalized for confidentiality), the improvement RESIN provides is significant. Both metrics show large decreases: VM reboots are reduced by 41× from September 2020 to September 2021, and allocation error rates are reduced by 10× from October 2020 to September 2021. Note that these key metrics have been continuously dropping before the starting points in Figures 9 and 10. We omit to plot these earlier points because the raw data are no longer in the database due to data retention policies.

The improvement also shows in the reduction of service incidents. In 2020 and 2021, no severe outages in Azure were caused by memory leaks (such outages occurred previously).

7.4 Effectiveness of Detection

Detecting memory leaks in a complex, frequently changing cloud infrastructure like Azure is challenging. RESIN aims to minimize the false positives and false negatives.

Precision To evaluate the detection accuracy, we count how many cases RESIN reported are flagged by developers as false alarms. Overall, RESIN only incurs 7 false positives out of 291 resolved cases in 14 months.

There are two common patterns of false positives: (i) after a component adds a new feature that consumes significantly more memory; (ii) after a configuration change, *e.g.*, *"we are collecting and correlating lot more counters and # of disks per node have also increased"*. The memory usages in these cases typically stabilize and become new baselines, which RESIN automatically picks up without requiring adjustment.

Recall We count how many cases RESIN misses or fails to detect in time. The criterion is a memory leak causing noticeable service impact and getting reported by users, developers, or other monitoring services before RESIN detects it. We search keywords including "memory leak" and "leak" on Azure's issue tracker used by all teams. We compare them with tickets automatically created by RESIN. Overall the false negatives are few: only 4 cases are not on RESIN's ticket list.

We also inspect the reason for each case. One case in July 2020 was caught by us manually when analyzing a heap snapshot trace, before the issue triggered RESIN's alert. In other two cases, the leak impact was not significant, but developers found the leaks when they were closely monitoring their new deployment. The last case was both found by developers and RESIN, but developers found the issue faster. This happened because the issue manifested itself earlier on a testing cluster that had a different workload from the production clusters RESIN monitored; developers were also using an aggressive threshold in that cluster to expose potential issues.

Timeliness It can be difficult to determine the exact starting time of a leak and hence the exact detection delay. We observe that RESIN's detection typically occurs within two hours of a leak's clear manifestation. In general, a too-long detection delay would be reflected in a false negative since developers or other monitoring tools would detect the issue earlier.

Resolution rate Not all cases are eventually resolved by developers. On average, RESIN reports a resolution rate of 52% within reported date range. We believe this resolution rate is underestimated compared to actual responsive rate: in many cases, developers took actions but did not update the open tickets (only tickets tagged with a high severity level are mandatory to resolve according to ticket platform's policy). "Unresolved" also does not necessarily mean the ticket is unimportant. We observed that some teams tend to have lower response rates, likely due to their limited resources and the overwhelming number of urgent tickets.

This result is also influenced by our design goal. RESIN is designed to catch memory leaks early before the leak escalates to catastrophic issues. RESIN further provides automatic leak mitigation. Thus, a could-have-been-severe leak would appear to be low-risk. When developers prioritize resolving high-impact tickets, it adversely affects the resolution rate for our leak tickets. Due to the large volume of reported cases from various groups, we could not afford to inspect all unresolved tickets and check with their owners.

7.5 Effectiveness of Diagnosis

The diagnosis module of RESIN in total collects traces and generates reports for 157 cases from July 2020 to August 2021 (Figure 8). For tickets related to kernel leaks or clusters with legacy OSes, RESIN is unable to collect traces. Before November 2020 the diagnosis module was in trial runs and diagnosis reports were not appended to tickets. We gradually enabled it for more clusters after the trial run was over.

Figure 11 shows how the ratio of tickets with collected traces (and diagnosis reports) increases, which correlates with

[2]In this paper, we only count those VM unexpected reboots and allocation errors caused by low memory and memory exhaustion.

```
virtual void AddDsmsCertificate(CertificateStore& ... {
-   for (; certHead != nullptr; certHead = certHead->Next) {
+   for (auto currentCert = certHead; currentCert != nullptr;
+        currentCert = currentCert->Next) {
-       if (certHead->Versions == nullptr)
+       if (currentCert->Versions == nullptr)
          continue;
-       auto latest = certHead->Versions->Latest;
+       auto latest = currentCert->Versions->Latest;
        if (latest == nullptr)
          continue;
        ...
    freeCertLst(certHead)
```

Figure 12: The fix for ServiceH leak.

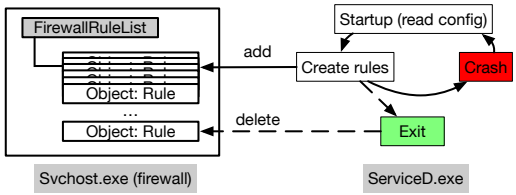


Figure 13: Contract violation induced leak on ServiceD.

the improvements on the ratio of resolved tickets.

Usefulness To evaluate the usefulness of the diagnosis reports RESIN generates, we randomly sample 14 issue tickets and closely follow up with the developers (all cases are eventually resolved and fixed). In 11 cases (79%), developers directly use the diagnosis reports to pinpoint root causes. Among them, in 5 cases the bug fix is in the same function in the allocation stack; in 5 cases the allocation stack and bug fix are in the same source file; in only 1 case the allocation stack and bug fix are in different components. Out of the other three cases, two are solved by memory dumps because the developers are quite experienced: upon seeing the sizes of leaked memory objects, they immediately realized which function the leaks came from. In one case the traces captured in the production cluster are not particularly useful. Instead, developers successfully captured some snapshots consisting of leaking stack on their own testing cluster.

Feedback Over time, developers build up high confidence in RESIN. We receive many pieces of positive feedback:

"(The result is..) incredibly useful. The information I had was enough."

"Thanks for pinpointing out the memory leak that we had been trying so hard to find over the past few days."

"Stack trace was sufficient for debugging this, it included the API call that was problematic."

Case studies We share two representative cases. The first case occurs in ServiceH[3]. This process' memory usage keeps increasing and gets restarted every few days. The diagnosis module in RESIN collects heap snapshots and pinpoints the root cause stack trace. After the diagnosis report is attached to the ticket, developers confirm and fix the issue in 3 hours.

In this case, the program uses a pointer to manage the list of certificates, and frees the pointer at the end of the function. However it also uses the pointer to traverse the list. In the end the pointer has moved and only a part of the list is freed (Figure 12). This is a day-0 bug introduced a long time ago, but is recently triggered due to added certificates to the machines.

[3]The service names are anonymized for reasons of confidentiality.

Mitigation	Count	50%	75%	90%	99%
Process restart	27,039	1.62	5.74	6.50	30.70
Kernel soft reboot	8,292	24.64	34.47	49.14	141.69
Node reboot	278,005	248.58	274.36	362.10	1382.61

Table 2: Single mitigation action execution time (seconds).

The second case represents another common (6 out of 14 cases we studied) type of leaks in cloud infrastructure: leaks due to contract violations in cross-component interactions. After RESIN reports a firewall-related `svchost` is leaking, the diagnosis module collects traces and reports a function in the rule list adding procedures after analyses.

Developers do not find bugs in this specific function at first, but the report prompts them to check the firewall rule lists on these machines. They then find the rule lists on these machines have been flooded with redundant rules. The reason is that the `svchost` process gets a firewall configuration from another program ServiceD. This program creates firewall rules at startup. Due to another bug, ServiceD keeps crashing, which causes it to miss deleting created rules and repeatedly recreate rules upon restarts (Figure 13). This in turn causes significant memory usage increases for the `svchost` program. Such a bug is hard to be detected statically.

Timeliness The diagnosis timeliness is also important to help developers. We measure the latencies of RESIN's heap snapshot collection and analysis. The median trace collection time is 61 minutes. For more than 80% of cases, the collection finishes within 10 hours. Note that the trace collection time is influenced by when a leak recurs in a suspected process. If the leak is sporadic, RESIN has to wait until the symptom reappears to capture the snapshot. For trace analysis, the median latency of the analysis jobs is 10 minutes.

7.6 Effectiveness of Mitigation

Mitigation procedure duration on leaked services Figure 14 shows the number of mitigated nodes of a kernel leak due to a buggy driver. At first, RESIN applied mitigation actions on a few nodes per day to test possible side effects. Once the mitigation actions reached production, RESIN applied mitigation to at most around 2,000 nodes per day with some fluctuations. The mitigation action volume then gradually dropped as the fix was being rolled out. Eventually the volume dropped to a few nodes a day, which were primarily nodes that failed in driver upgrading or other fix actions.

Mitigation action duration on single host We collect the frequencies and durations of each mitigation action between July 2020 to September 2021. As Table 2 shows, process restart is the most lightweight mitigation action. In most cases, it finishes within 6.5 seconds. Kernel soft reboot is also fast and in most cases finishes in a minute. Node reboot takes a longer time, with a median time of 4.6 minutes.

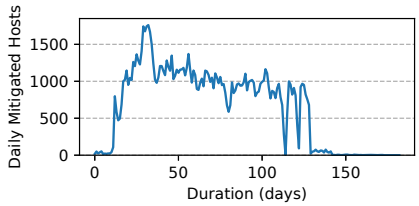


Figure 14: Mitigation for a leaking driver.

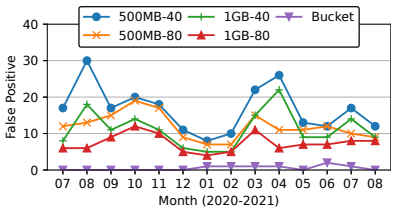


Figure 15: False positive of detection algo.

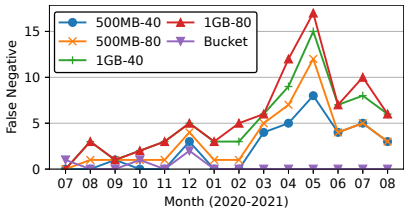


Figure 16: False negative of detection algo.

7.7 Comparison of Different Algorithms

Bucketization-based detection We first compare our core detection algorithm, the bucketization-based pivot analysis, with the practice of static threshold-based memory usage monitoring. We use four threshold policies, *e.g.*, policy "500MB-40" means generating leak alerts if a service's memory usage exceeds 500 MB on more than 40 nodes. We apply these hard thresholds to historical data and count how many cases will be wrongly reported as leaking (false positive) and how many leaking cases will be missed (false negative).

Figures 15 and 16 show the results. Our algorithm performs the best: it has both the lowest false positives and the lowest false negatives. In comparison, for other policies, it is often a dilemma to balance precision and recall. For example, policy "1GB-80" has the lowest false positives among the baselines at the cost of having the highest false negatives.

Pattern-based collection We compare our pattern-based collection with random collection. The experiment is conducted on ServiceS, ServiceV, and ServiceW. They have ongoing memory leaks on some hosts. We randomly choose six hosts and apply pattern-based collection on three hosts and random collection on the other three hosts. For the random strategy, we implement a workflow that periodically collects snapshots with at least two snapshots and completes the trace collection with a probability $1/6$. We inspect the collected heap snapshot traces to see if the leaking allocation exists in the snapshot.

Our pattern-based collection successfully captures leaking allocation stacks for all three services. Interestingly, the root cause of ServiceW was still unknown at the time we conducted the experiment. RESIN successfully captures an outstanding allocation that contains the bug within a real-time event processing function. In comparison, random collection only captures the buggy allocation for ServiceS, which has a frequent leaking interval (less than 1 hour).

Reference-assisted analysis We then evaluate the usefulness of reference snapshots with a controlled experiment on the ongoing leaking component ServiceS, which has the most noises among the three ongoing leak cases. We randomly sample eight hosts that have leaking patterns and collect snapshots until the leaking stack appears. We feed eight collected trace files to RESIN and compare the analysis results with and without reference snapshots. Figure 17 shows the result. Without the reference snapshots, the root cause stack trace ranks below the top three in all traces. With the reference snapshots, in 7 out of 8 traces, the root cause rank improves.

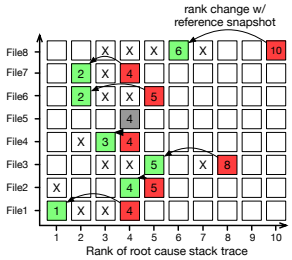


Figure 17: Ranks of root cause stack trace in diagnosis analyses on 8 trace files, w/ (green colored) and w/o (red colored) using reference snapshots. Cell with a number represents the rank. "X" marks the stack trace that gets filtered with the reference snapshots.

HasOverlap	Sessions	Nodes	25%	50%	75%	90%	95%	99%
FALSE	102,627	315	28	49	94	164	202	869
TRUE	165	31	38	50	59	86	241	888

Table 3: VM deployment time (seconds) impact by trace collection.

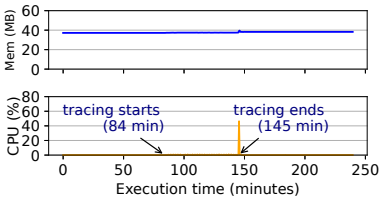
In four traces, the rank rises to the top three, which largely narrows down the code regions developers need to investigate.

7.8 Runtime Overhead

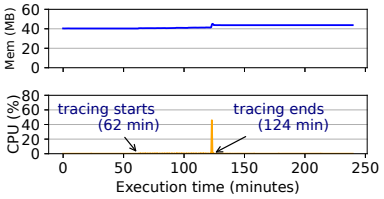
As a production service, RESIN should not impose significant overhead on the hosts. For the detection component, since RESIN leverages the kernel to collect performance counters infrequently and offloads the analyses remotely, the overhead is minimal. The main source of overhead is the heap snapshot trace collection. We use the VM deployment performance to quantify the end-to-end cost of trace collection, because VM deployment is the most important event for hosts and involves nearly all host services and triggers many critical code paths. A large overhead will be reflected in long deployment time.

We first check how many hosts RESIN performs trace collection on in November 2021. The result shows only 346 hosts are collected at least once, which is less than 0.1% of all nodes in a cluster. We then collect start and end timestamps of all VM deployment sessions and the heap snapshot tracing requests. We compare the timestamps in the two sets of events. In 315 (91%) of the 346 hosts, the deployment sessions do not have any overlap with tracing sessions, thus the tracing has no impact on these sessions.

Table 3 shows the end-to-end latency of the overlapped sessions compared to non-overlapped sessions: by 1 s for the median, and by 10 s for the 25th percentile. The latency increase could be notable for some short-duration deployments. However, this impact is limited to only a few sessions (0.16%) from a relatively small number of host nodes.



(a) host A



(b) host B

Figure 18: Memory size and CPU usage changes through tracing.

To measure the impact on memory size and CPU, we conduct experiments on two hosts that have active workloads. We trace one of the critical host processes. Figure 18 shows the memory and CPU usage during the experiment. Enabling tracing both slightly increases the average memory usage, 0.25 MB for host A and 0.53 MB for host B, and the CPU usage, 0.23% for host A and 0.22% for host B. When doing snapshot and dumping the trace, there is a clear spike for CPU usage: around 46% in both hosts. The memory usage also increases, but not significantly: 1.93 MB for host A and 3.89 MB for host B. After the tracing, the memory usage remains at a slightly increased level due to a one-time initialization required by tracer. Overall, the CPU and memory usage costs are acceptable in production deployment.

7.9 Tuning Effort

The software components and workloads in Azure infrastructure undergo frequent changes. As part of RESIN's design goals of minimizing false positives and false negatives, we aim to build robust algorithms that avoid fragile parameter tuning. For the detection part, throughout RESIN's production operation, we only made one major parameter change. We updated the alert threshold for the final result <*BucketSize*, *SeverityScore*> (Section 4.3.1) from <50 MB, 10> to <200 MB, 40> around 3 months after deploying RESIN. There has not been further tuning since then. For the diagnosis part, except for the initial trial runs in which we were experimenting with the snapshot algorithms, the parameters for the completion triggers have not been tuned after the diagnosis engine was enabled in production.

8 Lessons and Limitations

Lessons While memory leaks are generally taken seriously, developers tend to postpone the investigation if there are no convincing hints. Presenting clear evidence in results is critical, and significantly improves developers' responsiveness.

Many teams write extensive test cases that check if allocations are freed. Some teams also implement their own version of memory leak detection tool in their testing cluster. Developers mentioned a major pain point is that the testing environment has significant discrepancies with the production environment. For example, in one case, developers mentioned *"We don't have an environment where ServiceH runs for a really long time with hosts undergoing reboots."*, otherwise, their testing would have caught the memory usage anomaly.

Our initial thought in designing the diagnosis module is to analyze the source code of the detected leaking component. We later found that finding the root cause stack traces is usually good enough for developers to debug the issue based on their own experience and domain knowledge.

For production services, safety is of high priority. The cloud infrastructure is a complex and dynamic environment. Some workflow in RESIN can be interrupted abruptly, *e.g.,* due to transient network issues, interference with other profiling tools. On one occasion, RESIN accidentally left the trace collection running and triggered alarms in the detection engine. We set three lines of protection to prevent similar issues: (i) limit collecting on same cluster within one hour five times at maximum to reduce side effects; (ii) a forced cleanup operation whether the profiling succeeds or not; (iii) a workflow that periodically checks logs and cleans up for runaway hosts.

Limitations The telemetry data RESIN analyzes is relatively coarse-grained. Even the heap snapshot only contains limited information about allocations. Therefore, it has inherent inaccuracies and may miss detection of minor leak bugs. RESIN can be further enhanced by collecting more fine-grained signals, and leveraging semantic information from source code.

Developers may need to reproduce a reported memory leak issue for investigation or confirming bug fixes. But this is often challenging, because the issues are often triggered by complex workloads and rare conditions. RESIN does not address this challenge. We plan to automatically capture production triggering workloads for developers to reproduce leaks.

The patterns used in our heap snapshot trigger are based on empirical observations, which may be incomplete. Our classification method is simple. They can be improved with more comprehensive case studies and more advanced methods.

9 Related Work

Detecting memory leak bugs has been extensively studied in the context of conventional software. Our work focuses on addressing memory leaks in production cloud infrastructure, which face unique challenges as described earlier. Indeed, the memory leaks addressesed by RESIN are usually the ones that escape the bug detection and extensive testing practice in Azure and are only triggered in complex production workloads. The main research contribution of RESIN is its novel multi-stage approach and algorithms including the bucket pivot analysis and moving suspicious interval algorithm for leak detection, the live heap snapshot collection and analysis for leak diagnosis, and the decision tree based leak mitigation.

Dynamic leak detection. Many solutions have been proposed to dynamically detect memory leaks. There are broadly two approaches. In one approach, the tool records memory-related metadata by inserting checks to object code [16], using performance monitoring units in processors [24], instrumenting bytecode [39, 49], instrumenting intermediate representation [25, 34], instrumenting source code [21], modifying garbage collector or memory allocators [23, 35] or using metrics such as object staleness [17] as indicators to examine object lifetime. These works usually record fine-grained memory object information and have high accuracy, but they require rewriting codes or special hardware support, which is difficult and unsafe to apply in production settings.

Another approach analyzes heap snapshots/dumps [31, 33, 38, 44]. They are designed for interactive offline debugging and do not work well for long-running processes and services in production systems. They also rely on user-defined workloads as oracles to judge if memory growth is a leak. Obtaining such oracle workloads is difficult in practice. RESIN continuously monitors components in production cloud, designs robust algorithms to detect leaks without requiring oracles, and performs low-overhead live trace collection on-demand.

Some solutions [22, 40, 41] analyze memory usage patterns. They propose complex models to detect leaks in a single process or VM. The memory usage behavior of an individual process can be highly noisy due to workload effect and interference. Thus, they can have false positives and false negatives when applied in production cloud. Building complex models for each process in cloud scale also faces significant computation challenges. In comparison, RESIN focuses on the memory usage summary and global trend across processes, which enables accurate detection and efficient computation. RESIN additionally takes live heap snapshot and analyzes the snapshots to help developers localize the root cause.

Static leak detection. A wealth of work uses static analysis to find memory leak bugs. Many of them focus on improving the accuracies of static analyses [10, 15, 18, 36, 47]. Some other work focuses on finding specific leak code patterns. LeakChecker [50] finds objects created by the iteration are unnecessarily referenced by objects external to the loop. MLEE [46] finds leaks from early-exit paths by cross-checking the presence of memory deallocations on different early-exit paths and normal paths. Heapster [5] adopts a hybrid approach to leverage dynamic information to help static analysis. In general, while static approaches have the advantages of not requiring running a program, they face well-known scalability and accuracy challenges. They are also typically designed for a specific type of program. The software components in cloud infrastructure are highly complex and are written in a wide variety of programming paradigms. Also, static analyses cannot handle the forgotten leaks.

Leak fix and recovery. Some research work focuses on helping developers fix leak in addition to detecting them. LeakPoint [9] points developers to the potential fixable locations by taint analysis. LeakChaser [48] provides three layers of abstractions to assist programmers to diagnose memory leaks. Some other work focuses on automatically recovering the program from leaking. LeakSurvivor [42] and Melt [7] reclaim memory resources by swapping out objects to disks. LeakFix [11] inserts deallocations for leaks.

Statistical debugging. Statistical debugging [28, 29] uses statistical methods to identify predictors in the source code that correlate with a program failure. It requires instrumenting all predicates and re-running a program many times with normal runs and buggy runs. The diagnosis design in RESIN is complementary to statistical debugging. It collects live heap snapshots from production directly. Its algorithm identifies buggy stack trace based on the allocation information.

Failure detection and mitigation. Detecting memory leaks in production cloud is related to the topic of failure detection and mitigation in distributed systems [14, 19, 20, 27, 30, 43, 51]. Memory leaks are difficult to detect compared to other types of failures. IASO [37] detects fail-slow issues and supports mitigating slow issues with VM or node reboots. Narya [26] predicts node-level failures and performs mitigation actions. RESIN focuses on catching on-going memory leak issues, and provides a holistic solution. Its mitigation module leverages results from the detection engine to perform targeted mitigation to a specific process, service, driver, or host OS.

10 Conclusion

This paper presents RESIN, an end-to-end service designed to tackle memory leaks in production cloud infrastructure. RESIN takes a divide-and-conquer approach to decompose the memory leak problem, and designs a multi-level solution with novel algorithms including bucketization-based pivot analysis, live heap snapshot strategy, and diagnosis analysis. RESIN has been running in Azure for more than 3 years, and successfully reduces low-memory-induced VM reboots and new VM allocation errors by 41× and 10×, respectively.

Acknowledgments

We would like to thank our shepherd, Kathryn S. McKinley, and the anonymous reviewers for their thoughtful comments. We thank our colleagues who partnered with us on building the overall solution and providing feedback to us, including but not limited to Nathan Ernst, Anupama Vedapuri, Rui Ding, Carl Zhou, Francis David, Gaurav Jagtiani, Rakkimuthukumar Nallore Ponnusamy, Jayjit Phadke, Dustin Douglas, Matt Sebek, Harish Srinivasan, Kevin Broas, and Heejin Son. We would like to thank the strong support from Igal Figlin, Dongmei Zhang, Marcus Fontoura, Melur Raghuraman, Mark Russinovich, and Girish Bablani. This work was supported in part by the National Science Foundation grants CNS-1942794, CNS-2149664, CNS-1910133, and CCF-1918757.

References

[1] LeakSanitizer – clang 13 documentation. https://clang.llvm.org/docs/LeakSanitizer.html.

[2] Windows Performance Recorder. https://docs.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-recorder.

[3] Request for supporting LeakSanitizer. https://developercommunity.visualstudio.com/t/support-leaksanitizer/826620, 2019.

[4] Amazon. AWS service outage on October 22nd, 2012. https://aws.amazon.com/message/680342.

[5] M. Benz, E. K. Kristensen, L. Luo, N. P. Borges, E. Bodden, and A. Zeller. Heaps'n leaks: How heap snapshots improve android taint analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1061–1072, Seoul, South Korea, 2020.

[6] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, page 61–72, San Jose, California, USA, 2006.

[7] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, page 109–126, Nashville, TN, USA, 2008.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, NSDI '05, page 273–286. USENIX Association, 2005.

[9] J. Clause and A. Orso. LEAKPOINT: Pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, page 515–524, Cape Town, South Africa, 2010.

[10] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 72–82, Montreal, Quebec, Canada, 2019.

[11] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for C programs. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, page 459–470, Florence, Italy, 2015.

[12] M. Ghanavati, D. Costa, A. Andrzejak, and J. Seboek. Memory and resource leak defects in java projects: An empirical study. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 410–411, Gothenburg, Sweden, 2018.

[13] M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak. Memory and resource leak defects and their repairs in java projects. *Empirical Software Engineering*, 25(1):678–718, 2020.

[14] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Oakland, CA, USA, 2018.

[15] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 310–323, Long Beach, California, USA, 2005.

[16] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*, page 125–138, Berkeley, 1992.

[17] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '04, page 156–164, Boston, MA, USA, 2004.

[18] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 168–181, San Diego, California, USA, 2003.

[19] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[20] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI. ACM, May 2017.

[21] S. H. Jensen, M. Sridharan, K. Sen, and S. Chandra. MemInsight: Platform-independent memory debugging for javascript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, page 345–356, Bergamo, Italy, 2015.

[22] A. Jindal, P. Staab, J. Cardoso, M. Gerndt, and V. Podolskiy. Online memory leak detection in the cloud-based infrastructures. In *International Conference on Service-Oriented Computing*, ICSOC '20, pages 188–200, Dubai, United Arab Emirates, 2020.

[23] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 31–38, Nice, France, 2007.

[24] C. Jung, S. Lee, E. Raman, and S. Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, page 825–836, Hyderabad, India, 2014.

[25] S. Lee, C. Jung, and S. Pande. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of the 36th International Conference on*

Software Engineering, ICSE '14, page 814–824, Hyderabad, India, 2014.

[26] S. Levy, R. Yao, Y. Wu, Y. Dang, P. Huang, Z. Mu, P. Zhao, T. Ramani, N. Govindraju, X. Li, Q. Lin, G. L. Shafriri, and M. Chintalapati. Predictive and adaptive failure mitigation to avert production cloud vm interruptions. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, OSDI '20. USENIX, November 2020.

[27] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati. Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20. USENIX, February 2020.

[28] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 15–26, Chicago, IL, USA, 2005.

[29] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, oct 2006.

[30] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 559–574. USENIX Association, Feb. 2020.

[31] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, page 115–124, Washington D.C., USA, 2010.

[32] Microsoft. Windows kernel api: ExAllocatePoolWithTag. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-exallocatepoolwithtag.

[33] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, ECOOP '2003, pages 351–377, Darmstadt,Germany, 2003.

[34] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, San Diego, California, USA, 2007.

[35] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 397–407, Dublin, Ireland, 2009.

[36] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Proceedings of the 13th International Static Analysis Symposium*, SAS '06, pages 405–424, Korea, 2006.

[37] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi. IASO: A fail-slow detection and mitigation framework for distributed storage services. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 47–61, Renton, WA, USA, 2019.

[38] W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, page 116–134, Genova, Italy, 1999.

[39] D. Rayside and L. Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 194–203, Atlanta, Georgia, USA, 2007.

[40] V. Šor, P. Oü, T. Treier, and S. N. Srirama. Improving statistical approach for memory leak detection using machine learning. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 544–547, Washington, DC, USA, 2013.

[41] V. Šor and S. N. Srirama. A statistical approach for identifying memory leaks in cloud applications. In *Proceedings of First International Conference on Cloud Computing and Services Science*, CLOSER '11, pages 623–628, Noordwijkerhout, Netherlands, 2011.

[42] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *Proceedings of the 2008 USENIX Annual Technical Conference*, USENIX ATC '08, pages 307–320, Boston, MA, USA, 2008.

[43] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: Understanding the causes and impact of network failures. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 315–326, New Delhi, India, 2010.

[44] J. Vilk and E. D. Berger. BLeak: Automatically debugging memory leaks in web applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, page 15–29, Philadelphia, PA, USA, 2018.

[45] C. A. Waldspurger. Memory resource management in vmware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02. USENIX Association, Dec. 2002.

[46] W. Wang. MLEE: Effective detection of memory leaks on early-exit paths in OS kernels. In *Proceedings of the 2021 USENIX Annual Technical Conference*, USENIX ATC '21, pages 31–45, July 2021.

[47] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '13, page 115–125, Lisbon, Portugal, 2005.

[48] G. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 270–282, San Jose, California, USA, 2011.

[49] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 151–160, Leipzig, Germany, 2008.

[50] D. Yan, G. Xu, S. Yang, and A. Rountev. Leakchecker: Practical static memory leak detection for managed languages. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 87–97, Orlando, FL, USA, 2014.

[51] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan. *Understanding and Detecting Software Upgrade Failures in Distributed Systems*, page 116–131. Association for Computing Machinery, New York, NY, USA, 2021.

Cancellation in Systems

An Empirical Study of Task Cancellation Patterns and Failures

Utsav Sethi
University of Chicago
usethi@uchicago.edu

Haochen Pan
University of Chicago
haochenpan@uchicago.edu

Shan Lu
University of Chicago/Microsoft
shanlu@uchicago.edu

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Suman Nath
Microsoft Research
Suman.Nath@microsoft.com

Abstract

Modern software applications rely on the execution and coordination of many different kinds of tasks. Often overlooked is the need to sometimes prematurely terminate or cancel a task, either to accommodate a conflicting task, to manage system resources, or in response to system or user events that make the task irrelevant. In this paper, we studied 62 cancel-feature requests and 156 cancel-related bugs across 13 popular distributed and concurrent systems written in Java, C#, and Go to understand why task cancel is needed, what are the challenges in implementing task cancel, and how severe are cancel-related failures. Guided by the study, we generalized a few cancel-related anti-patterns and implemented static checkers that found many code snippets matching these anti-patterns in the latest versions of these popular systems. We hope this study will help guide better and more systematic approaches to task cancellation.

1 Introduction

Task cancellation is critical to the performance and availability of modern concurrent and distributed systems. Unlike fault handling, which reacts to the failure of a software or hardware component, task cancellation proactively stops the execution of a software component (i.e., a task) that no longer needs to run. Concurrent applications use task cancellation for better resource management, task coordination, and system responsiveness [6, 7, 20, 22]. For instance, when a user aborts a long-running operation, the underlying system may want to cancel the relevant tasks to save resources; when a high-priority request comes, a busy system may want to cancel a low-priority task for the greater good. Task cancellation is crucial for today's systems that concurrently execute a large number of complex and resource-consuming tasks under stringent quality of service requirements.

Unfortunately, supporting efficient and correct task cancellation in modern applications is nontrivial. Tasks need to be designed such that they can be aborted at certain points of execution without undesirable side-effects (e.g., without corrupting the system state). Moreover, the application needs to decide when to safely cancel a task, and once decided to cancel, the decision needs to be correctly propagated to the target task to be canceled.[1] Last but not least, a system may contain dozens or hundreds of concurrent tasks, with complex dependencies among the tasks as well as on the system environment. If not carefully implemented, canceling a task may break a dependency or introduce concurrency errors such as races. It is therefore not surprising that implementing task cancellation can be error-prone.

As it stands, there have been no studies on task cancel problems in concurrent and distributed systems—how cancel is used and implemented, the various types of cancel-related bugs, the impact of those cancel-related bugs, and so on, although various other types of bugs and problems have been heavily studied for distributed systems [11, 16, 18, 26, 27].

This paper attempts to provide an in-depth analysis of cancellation usage and problems in popular software applications across multiple languages, which we hope will help guide cancellation-related systems research and design.

Why do applications cancel tasks? To understand why cancellation may be desirable to system operation, we reviewed 62 *feature requests* in 13 popular open-source applications, such as HBase, Hive, Cassandra (Java); Roslyn, ASP.NET Core (C#); CockroachDB, and InfluxDB (Go).

We found that about half of the cancel-feature requests aim to terminate tasks that no longer produce useful results upon a change in system or user state (e.g., the finish of a related task and the end of a user session); close to half of the requests aim to improve operational flexibility and enable users to cancel a job, particularly the time-consuming ones, at any time; a small number of requests aim to enable stopping a low-priority task prematurely to support the launching and running of other more important tasks.

Our study confirms our understanding that task cancellation is a crucial feature that facilitates efficiency and operation flexibility in concurrent systems. It shows that the trigger of a cancel can be a variety of events (far beyond

[1] This is in contrast to fault-handling where the external environment decides *when* a fault is generated.

	Task	Task Cancellation
C#	`Task,Thread`	`CancellationToken` struct
Go	`goroutine`	`Context` type
Java	`Thread`	`interrupt()` on `Thread` itself

Table 1. Task constructs and cancellation mechanisms

system shutdown and component failures), and the target of cancellation is often a small number of selective tasks (rarely bulk cancellation), which can all bring complexity to the implementation of task cancellation.

What causes cancel-related bugs? To understand the challenges in implementing task cancellation correctly, we studied 156 bug reports across the same set of 13 popular open-source applications in Java, C#, and Go to understand what are common cancellation-related bugs.

Our study shows that problems routinely occur at all phases of cancel: 1) deciding when and which task to cancel (about one third of the bugs), 2) propagating the cancel request from the initiator to the target task (about one quarter of the bugs), and 3) fulfilling the cancel in the target task (about one third of the bugs). Some classes of problems are particular to the type of mechanism used to issue cancel, such as bugs in the use of Java's `interrupt` API, and bugs in passing cancellation tokens through function parameters in C# and Go. Many other classes of problems are due to the overall complexity of implementing cancel, such as determining which tasks conflict, which system state changes must be reverted before task termination, etc. For each type of bugs, we discuss potential solutions to tackle them.

Impacts of cancel-related bugs. The impact of cancel bugs varies, but can in some cases be severe. Among issues with specified symptoms, a few common categories are resource leaks, performance issues, broken task APIs, data corruption or loss, and incorrect user reporting.

Cancellation anti-patterns. Through the study above, we have generalized and implemented static checkers for five cancel-related anti-patterns using the CodeQL [1] static analysis framework, including (1) missing interrupt handling inside a loop (Java); (2) using the wrong built-in API to check or reset the interrupt flag on threads (Java); (3) failure to propagate cancel to child tasks (Java); (4) ignoring cancel-token parameters (C#); and (5) not propagating cancel tokens (C#)[2]. We find around 200 instances of these anti-patterns across the latest versions of the 13 applications we studied, which further motivates future work to improve the support for correct cancel implementation.

2 Background

Task. This paper defines a task as a unit of concurrent execution. As summarized in Table 1, in Java, all code that implements a `Runnable` interface qualifies (e.g., `Thread`). In C#, tasks are objects of type `Thread` or `Task`. In Go, execution inside a `goroutine` is a task [4, 5, 22]. Tasks are not limited to any specific programming model: for example, some issues we study involve tasks as part of an event-driven design. Some tasks execute with a clear end, like a user-request task launched by a server application; some execute with an open end and cease only on system shutdown or explicit request to terminate, like a task that provides an in-memory cache service for others. Tasks can also initiate work on other nodes, e.g. by issuing an RPC call.

[2]This particular checker is a re-implementation of an existing C# checker.

```
1  public void run() {
2       try { ...
3       } catch (InterruptedException e) {
4       // receiver handles the cancel request
5       }
6       ...
7       if (Thread.currentThread().isInterrupted()) {
8       // receiver handles the cancel request
9       }
10   }
```

Listing 1. Handling cancel requests in Java

Task Cancel. Cancel is the deliberate attempt of one task to terminate another task in a *cooperative* way. We will refer to the former as the cancel initiator and the latter as the cancel target. All the instances of cancel we study are *cooperative*, which means that the target task, upon receiving the request, chooses how and when to terminate [15]. Note that the alternate way of task cancel - *abortive*, where the initiator forces the target to terminate - is prone to semantic errors and is not supported by the three languages that our study focuses on (Java, C#, Go). For example, the abortive Java `Thread.stop()` method is deprecated now.

Cancel vs Fault Handling. Task cancel and fault handling have some similarities in that they both involve a task finishing earlier than expected, but they also have fundamental differences. Cancel can be considered part of the regular operation of the system: the conditions that cause cancel to be issued are known and expected with some regularity, such as to proactively prevent performance problems, as we will discuss in Section 4; the cancel process involves the cooperation between at least two running parties, the initiator and the target; after the cancel is conducted, the system is expected to remain functioning as normal or even at a higher capacity. This is in contrast to failure handling, in which failure events are unexpected; the handling is reactive after a component failure; and the expectation for system functioning may be lower - e.g. to function at reduced capacity, or to terminate safely.

Cancel mechanisms. Although the built-in cancel mechanisms in C#, Go, and Java take different forms, as listed in Table 1, they all essentially offer a "flag": the initiator sets the flag when requesting cancel, and the target can check the flag and respond to the cancel request.

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;
var mytask = Task.Run(() => {
// the receiver checks the token before starting
// to handle a potential cancel request
    ...
    if (token.IsCancellationRequested) {
    // receiver handles the cancel request
    }
}, token);
```

Listing 2. Handling cancel requests in C#

Specifically, in Java, any thread can execute `t.interrupt()` to set an internal flag of thread `t`. Any code executing in thread `t` can use APIs like `isInterrupted()` to check this flag and see if an cancel request has been delivered to it. Alternatively, any execution of a blocking API, like `sleep()` or `poll()`, will throw an `InterruptedException` upon the setting of its thread's cancel flag, as shown in Listing 1.

C# and Go offer more flexible ways of cancel. Instead of limiting each thread to have one flag, they allow the software to declare any number of `CancellationToken` structs (C#) or `Context` variables (Go) that each contains a cancel flag. In C#, a `CancellationToken` object, generated from a `CancellationTokenSource` is typically passed through function parameters. An invocation of `Cancel()` on the token's source would set the flag inside the token object, which is visible to any task that has access to the token, as illustrated in Listing 2. Cancel in Go is similar: the `Context` type provides a `CancelFunc` to issue a cancel signal, which can be checked via ctx.Done() on the Context ctx. Like `CancellationToken`, `Context` is typically passed via function parameters. In the remainder of the paper, we will refer to `Context` variables also as cancellation tokens for simplicity.

The `CancellationToken` in C# also allows registering a callback function to be called when the token is canceled. This functionality is rarely used in the applications that we study and hence will not be discussed in this paper.

Finally, developers can implement custom means of cancel. In many Java programs, shared Boolean variables are used as cancel flags. Threads explicitly read and write these flags to carry out cancel. This essentially allows multiple cancel flags for one thread and hence can embed more semantic information inside each flag. However, it is also prone to bugs, as we will discuss in Section 5.

3 Methodology

Application selection. We study applications written in three different languages: Java, C# and Go, as shown in Table 2. These languages were chosen as they have widespread use of different built-in cancel mechanisms, and as such provide a useful point of comparison for this study.

In choosing which Java applications to study, we focus primarily on the most popular, as indicated by GitHub stars, open-source distributed applications in various categories,

Table 2. Applications included in our study

Application	Category	Stars	Bugs	CFR[2]
Java (distributed apps)				
Cassandra	Database	7K	14	2
Elasticsearch	Full-text search	57K	15	20
Hadoop[1]	Distri. storage; distri. processing	12K	10	3
HBase	Database	4K	26	3
Hive	Data warehousing	4K	21	5
Kafka	Stream processing	20K	9	2
Solr/Lucene	Full-text search	4K	9	2
Spark	Data processing	31K	6	6
Java - subtotal			110	43
C# (single-instance apps)				
ASP.NET Core	Web framework	26K	6	1
Roslyn	Compiler	15K	14	8
C# - subtotal			20	9
Go (distributed apps)				
CockroachDB	Database	22K	12	6
etcd	Key-value store	38K	8	0
InfluxDB	Database	22K	6	4
Go - subtotal			26	10
Total			156	62

[1] Including Hadoop Common, HDFS, YARN, MapReduce
[2] Cancel-Feature Requests

as listed in Table 2. Our selection is more limited for Go and C#, since there are much fewer applications written in these two languages on GitHub. For Go, we study applications that are analogous to categories studied in Java: InfluxDB and CockroachDB (distributed databases), and etcd (distributed application serving and coordination). For C#, there do not exist any widely-used applications in those categories. So, as an alternative, we chose the top 2 applications/frameworks, out of the 50 most popular C# applications on GitHub, that utilize cancel extensively: Roslyn (compiler suite) and ASP.NET core (web framework).

Cancellation Issue Study. For these selected applications, we checked their Jira issue trackers or GitHub issue-and-pull systems, if they do not use Jira. We searched for *resolved* and *valid* issues, up to June 2021, using the following keywords: *abort*, *cancel*, *interrupt*, and *terminate*. We then manually checked the reports to exclude issues that do not have a clear description or are unrelated to task cancel.

From the remaining, we get 156 issues that are labeled by developers as "bug" or are clearly fixing a bug, although not labeled. They will help us understand the root causes and symptoms of cancel-related bugs, as presented in Section 5 and 6. We should note that although an issue might belong to multiple root causes or symptom categories, it

Table 3. Reasons underneath Cancel-Feature Requests (CFR)

Why should a task T be canceled?	#CFR
A. Efficiency: T no longer produces useful results	**30**
- A1. Upon system shutdown	5
- A2. Upon a user disconnection or time-out	6
- A3. Upon a system or user event	19
B. Flexibility: T is no longer wanted by users	**28**
- B1. Cancel through an API call	20
- B2. Cancel through user interface or keyboard	7
- B3. Cancel through timeout parameter	1
C. Priority: More important tasks need to run	**4**
Total	**62**

is classified by its primary category only, **without double-counting**. In addition, we study 62 issues that are requests to add the capability of canceling some tasks and are labeled as "improvement" or "feature", instead of "bug", and contain patches approved or already merged. They will help us understand the motivation of task cancel, as in Section 4.

We believe cancel problems are under reported, as cancel code can be difficult to exercise during testing. From the discussion in cancel-feature requests, we also see that the complexity in correctly implementing task cancel sometimes drives developers away from implementing cancel, which of course comes with performance and efficiency loss.

Threats to validity. Our study does not cover all task cancel mechanisms, and may not generalize to those issues and systems not covered in our benchmark suite. Particularly, we have skipped those cancel-feature requests and cancel-related bugs whose description is not clear enough for us to conduct further categorization. We may also have missed cancel-related requests or bugs whose reports do not contain the search keywords used by us. Furthermore, since there are many more issue reports and pull requests about adding cancel features than those about cancel-related bugs, we limit our study of cancel-feature requests to those that contain cancel-related keywords in the issue/pull titles. Thus, we likely have missed many requests that have those keywords in the issue/pull body, but not the title.

4 Why Do Applications Cancel Tasks?

To understand why tasks may require cancel and what triggers a task cancel, we studied 62 cancel-feature requests in Java, C#, and Go systems, following the methodology described in Section 3, and generalized three main reasons for task cancel as shown in Table 3.

Reason-A: Efficiency. Close to half of the cancel-feature requests originate from developers' efficiency concerns, as the computation of a task T no longer produces useful results upon (A1) a system shut-down, (A2) a user-session termination, or (A3) a particular system or user event. Among these three different cancel-trigger scenarios, A3 is the most common and triggers cancel at a finer granularity than A1 and A2. For example, when a user navigates away from a web page P, the system still runs many tasks related to the user, but can cancel all the tasks initiated by page P (e.g., influxdb-19029); when one attempt of a task finishes, all other speculative or parallel attempts of this task can be canceled (e.g., SPARK-25773 and roslyn-8050); when a job is canceled or finished, its related tasks can be canceled (e.g., roslyn-25620 and roslyn-51816). In all these cases, continuing the execution of T does not affect functional correctness but wastes system resources and affects request latency.

Reason-B: Flexibility. Another common reason is to offer users the flexibility to prematurely terminate a user operation and all its related tasks, which contribute to about 40% of the cancel-feature requests. In a number of cases, the requests explicitly mention that the target task may take a long time (e.g., elasticsearch-72644 and elasticsearch-73818 and SOLR-6122) or even hang for unknown reasons (e.g., KAFKA-1506), and hence should be cancellable. In other cases, the exact reasons why a user may want to cancel a task is not explained. The requested cancel features typically get implemented as task-cancel commands or as handlers of certain user interface events, like the Ctrl+C keyboard combination.

Reason-C: Priority. Interestingly, sometimes, developers want to enable the system to sacrifice T for the benefit of other more important tasks. For example, in HDFS-2507, a feature is added to cancel an ongoing checkpoint task of a standby NameNode when the active NameNode fails. This would allow the standby NameNode to immediately start the fail-over task instead of waiting for the long checkpointing to finish, minimizing the system downtime. Similar decisions of sacrificing long-running low-priority tasks for the benefit of high-priority tasks also occur in other systems (e.g., CASSANDRA-14397, elasticsearch-56009).

Observations. *Trigger variety.* A task cancel can be triggered by a variety of events, as shown in Table 3. This variety adds complexity to the implementation of cancel: the program may miss a trigger and fail to initiate the cancel. Even when a trigger is sensed, the trigger information may not be included in the cancel request, e.g., in Java's built-in cancel mechanism, making it difficult for the cancel handler to process the cancel request properly.

Fine granularity. Task cancel is often targeted; bulk cancel scenarios like system shutdown are rare. This fine granularity can make it difficult to decide which task to cancel.

Heavy coordination. In a system that involves many concurrent components, cancel may involve a lot of coordination across tasks: a task's cancel could be due to the launch, the progress, or the termination of another task. This heavy coordination requirement demands careful synchronization and shared-state clean-up during task cancel.

Proactive instead of reactive. Unlike fault handling, task cancel rarely reacts to an already exposed component failure. It is more about the system efficiency, request latency, operational flexibility, and resource balancing, which, although do not immediately precipitate system outages, are crucial to the service quality and robustness.

5 Root Causes of Cancel-Related Bugs

We divide the whole procedure of cancel into three phases, and categorize cancel bugs' root causes accordingly:

1) *Initiating Cancel* - the cancel initiator senses a cancel-trigger event and decides which task to cancel.

2) *Propagating Cancel* - the cancel request propagates from the initiator to the target.

3) *Fulfilling the Cancel* - the cancel target responds to the cancel request, releasing resources, restoring system states, and ending its own execution.

Note that there are 9 bugs caused by miscellaneous semantic errors that are not related to the core functionality of task cancel. We put them in the "Other" category in Table 4 and skip discussion about them below.

Table 4. Cancel-related bugs: root causes

Root Cause Category	Java	C#	Go
Buggy cancel initiation			
- Overlooking triggers	22	3	9
- Broken trigger checking	7	0	0
- Excess cancel	7	1	0
Buggy cancel propagation			
- Untimely delivery	15	3	4
- Dropped cancel	17	5	2
Buggy cancel fulfill			
- Cancel not checked	8	0	4
- Cancel not carried out	6	0	0
- Defective cleanup	23	5	6
Other	5	3	1

5.1 Cancel-initiation bugs

As discussed in Section 4, a variety of conditions might trigger a cancel. Deciding when to initiate a cancel to which target task is complex and susceptible to problems, contributing to about 30% of cancel-related bugs (Table 4).

In some cases, a cancel is not initiated when it should be, either because the system completely overlooks a cancel trigger ("Overlooking triggers") or because the system checks the existence of a cancel trigger incorrectly ("Broken trigger checking"). In other cases, a cancel is incorrectly or unnecessarily initiated ("Excess cancel"). We describe each type in more detail below.

5.1.1 Overlooking triggers. This type of bug occurs when a cancel should be initiated upon a specific trigger, but no logic exists to do so. This is the most common type of cancel-initiation bug, contributing to more than 20% of all the cancel-related bugs.

The most common scenario is that a running task T is canceled or has failed but a dependent task, which is no longer necessary, is not canceled. As an example, in SPARK-21738, expensive jobs would continue to run on a Spark cluster even after a user session was closed, wasting computation resources to produce irrelevant results. While Spark does provide support for canceling jobs, the system did not realize that a session closure should be treated as a trigger for job cancel.

As another example, in roslyn-1086, the failure of a compilation task will prevent a "completion" event from ever being published to an event queue, while a task listening to the queue, *AnalyzerDriver*, will continue to run and wait for the event which will never arrive. The solution in this case was to include a reference to the *AnalyzerDriver* in the compilation task, which is canceled via cancellation token upon compilation failure.

Other types of triggers could also be overlooked. For example, in CASSANDRA-8805, developers realized that the launch of high-priority tasks like *repair* often gets blocked by long-running low-priority tasks like *index-summary redistribution*, as these tasks access *sstables* in a conflicting way and cannot run in parallel. To solve this problem, developers added the logic to allow any *repair* to check for and cancel any running *index-summary redistribution* tasks.

Note that bugs of this type share similar root causes with those cancel-feature requests for efficiency or priority reasons, which were discussed in Section 4. The difference seems to be the impact: the ones that cause more severe failure symptoms are reported as bugs, instead of feature requests.

The patches to these bugs are straightforward: adding the logic to initiate a cancel upon the occurrence of the trigger.

Lessons learned. A fundamental challenge here is to track the dependency relationship among all the concurrent tasks, a daunting task in modern concurrent and distributed systems: which tasks conflict with each other and cannot run in parallel; which tasks depend on which task and hence should not continue if the latter is canceled; which tasks are redundant copies of which task and hence should not continue if the latter finishes successfully; etc. In all systems that we have checked, this is conducted in an ad-hoc way. There is an unmet need for coherent tool/framework and possibly programming language support for capturing these dependencies.

One particular type of dependency, the parent-child relationship, is feasible to track through static program analysis. Consequently, we can build a static checker to automatically identify code snippets where the parent task is canceled, and

yet no cancel is initiated towards the children tasks. We will present more details about this checker in Section 7.3.

Other types of dependencies, like *repair* versus *index-summary redistribution* or a speculative task versus the original task, depend on application-specific semantics and are much harder to track systematically. We noticed that these semantic-rich dependencies are often centered on some key shared data, like the `sstables` that are updated by conflicting tasks or the common `job-ID` shared between multiple job attempts (e.g., HIVE-12307). Consequently, future work may automatically infer task dependencies by analyzing access patterns on key data.

5.1.2 Broken trigger checking. Sometimes, the program anticipates the existence of a trigger. However, it checks the trigger occurrence in a wrong way. For example, in SOLR-10525, if a duplicate task is submitted while a previous instance of a task is still running, the previous instance should be canceled. However, the logic to recognize whether a previous instance of a task is running is incorrect and so a cancel is never issued, leading to the execution of duplicate tasks.

Lessons Learned. Many bugs of this type are related to checking whether a particular task is running. Often, the task performing the check does not have a direct reference to the task under check, and hence needs to refer to an intermediary, like a shared collection of task status. The logic to store and retrieve the task status information is custom implemented in each system and hence prone to bugs: some accesses to the task registry are not thread safe; different types of tasks may store their information in different ways in the collection and hence got mis-checked later; etc. Some standard library support would help.

5.1.3 Excess cancel. Converse to "Overlooking triggers", sometimes triggers are correctly sensed and yet tasks are wrongly or unnecessarily canceled. For example, upon the launch of a task T, the software may incorrectly cancel tasks that are actually not conflicting with T (CASSANDRA-13142, CASSANDRA-15024) or tasks that are indeed conflicting but have higher priority than T (HBASE-17674). Upon the finish of a task T, the software may incorrectly cancel tasks which are related to T but whose results are still needed (roslyn-11470, HADOOP-6762).

Lessons Learned. Similar as "overlooking triggers", these bugs originate from the challenge of tracking the dependency among tasks. Future research should study how to track which tasks conflict with or depend on each other, potentially through data dependency analysis.

5.2 Cancel-propagation bugs

Once a cancel trigger is correctly sensed and the cancel target is correctly identified, the initiator issues a cancel request. For about a quarter of the cancel-related bugs in our study, the propagation from the initiator to the target went wrong.

```
1  // Cancel initiator
2  class Initiator {
3    Task myTask;
4    main() {
5      ...
6      myTask.cancelFlag = true;
7    }
8  }
9
10 // Cancel recipient
11 class Task {
12   public boolean cancelFlag = false;
13   private BlockingQueue Bqueue;
14
15   run() {
16       while(cancelFlag == false) {
17         ...
18         Bqueue.take(); // blocks until an element is
                available
19       }
20   }
21 }
```

Listing 3. An example of late cancel (SPARK-1582)

5.2.1 Untimely delivery. It is important that a cancel can be issued at any time to the cancel target without delays or mis-handling. However, this is often not the case when a custom cancel mechanism is used.

Cancel race. In many systems, a "task manager" is implemented to coordinate tasks and relay cancel requests: the cancel initiator notifies the task manager about its cancel request; the task manager then sends the request to the cancel target. In several Java and Go systems, such as Cassandra (CASSANDRA-9070), Spark (SPARK-4097), HBASE (HBASE-13146), InfluxDB (influxdb-9018), and etcd (etcd-8443), the implementation of task managers contain concurrency bugs that manifest when cancel is issued at a special moment, like shortly after the target task is submitted, or in parallel with another cancel request towards the same target. As a result of these bugs, cancel requests may be dropped.

Occasionally, such cancel-related concurrency bugs also occur when a standard cancel mechanism is used. For example, in aspnetcore-11757, a cancel initiator disposes a `CancellationTokenSource` right after it requests a cancel on the token. As a result, when the target task checks the token, a use-after-disposal error occurs.

Lessons Learned. It is alarming that similar cancel-concurrency bugs occur in so many different systems. On one hand, standard task-manager library support could help. On the other hand, existing concurrency bug detection and testing tools [10, 13, 14, 17, 19] should be applied to check the correctness of cancel-related implementation.

Late polling. As discussed in Section 2, many custom cancels are conducted through a shared flag variable. Unfortunately, without system support, such a cancel request cannot be delivered timely when the target task conducts frequent blocking operations. For example, Listing 3 illustrates a simplified version of bug SPARK-1582. A task checks

```
// Cancel recipient
class Task {
  run() {
      ...
      commitSync() // interrupt lost inside commitSync
      ...
      if (isInterrupted()) {
        // cleanup steps here will not be performed
      }
  }
  commitSync() {
   sleep(1000); // unsets interrupted flag
    ...
    catch (InterruptedException ex) {
      // does not reset flag, cancel gets dropped
     }
  }
}
```

Listing 4. An example of dropped delivery (KAFKA-4375)

```
class Task {
  ...
  void checkStale() {
       ...
    // current thread is interrupted somewhere
     } catch (InterruptedException e) {
-      Thread.currentThread().interrupted(); // Wrong
+      Thread.currentThread().interrupt(); // Fixed
     }
  }
}
```

Listing 5. API Misuse Example (SOLR-8066)

whether a cancel is delivered to it at the beginning of every work-loop iteration through a custom `cancelFlag` variable. Unfortunately, since every iteration of the loop executes a `BlockingQueue::take()` operation, the flag may not be checked for a long or even unlimited amount of time, causing severe delays in Spark job cancellation. Similar issues also exist in KAFKA-5697, KAFKA-5896, and others.

These problems are typically fixed by using a language built-in cancel mechanism instead of, or in addition to, the custom flag to carry out the cancel. In Java, the built-in `Thread.interrupt()` would terminate blocking operations such as `sleep()`, `BlockingQueue::take()`, and `poll()`, with an `InterruptException` thrown. In C# and Go, many system operations such as `sleep()` accept cancellation tokens as parameters, allowing the timely delivery of cancel.

Lessons Learned. The key takeaway here is to avoid using a custom cancel flag, particularly when the nearby code region conducts blocking operations. We can use static program analysis to identify these vulnerable custom-cancel loops and warn the developers. Having said that, the pervasive use of custom-cancel loops in Java programs is probably due to the limitation of Java's built-in cancel mechanism, which we will discuss more in Section 5.4.

5.2.2 Dropped cancel.

Depending on the different cancellation mechanisms, a cancel request could be dropped before it propagates to the right target in different ways.

Cleared interrupt (Java). A tricky aspect of Java's built-in mechanism is that the interrupt received by a thread can be silently unset by methods along the call chain. As a result, the interrupt may fail to reach the code that is prepared to fulfill the cancel request, contributing to about 15% of cancel-related bugs in Java programs in our study.

For example, in KAFKA-4375, function `run` contains a well written cancel handler that stops child tasks and exits. Unfortunately, at run time, the cancel is often intercepted by the `sleep` method inside its callee `commitSync`, as shown in Listing 4. The Java `sleep` method, just like many other Java blocking methods, silently unset the interrupt and throw an `InterruptedException`. Without rethrowing the exception or resetting the interrupt flag, the interrupt is dropped before reaching the right handler in function `run`. Similar problems also occur in other systems, like HBASE-5243, HIVE-13858, HBASE-10650, HBASE-10651, HBASE-10652, etc. Patches for these bugs simply re-throw the interrupt in the catch block.

A related mistake is that developers sometimes get confused about a few similar Java APIs: `t.interrupt()` interrupts a thread `t`; `t.interrupted()` checks whether `t`'s interrupt flag is set *and* clears the flag; `t.isInterrupted()` conducts the same checking but *does not* clear the flag. When `interrupted()` is mistakenly used, the cancel could be dropped before reaching the intended cancel handler, as illustrated in Listing 5. This type of mistake occurred at multiple places across different systems (KAFKA-9415, KAFKA-5665,HBASE-10455, SOLR-8066). Patches for these problems are straightforward, as shown in Listing 5.

Lessons Learned. Many bugs of this type can be automatically detected. As we will discuss in Section 7.1 and 7.2, static checkers can search for the catch blocks of `InterruptedException` that neither terminate the execution nor re-throw the exception, and search for incorrect use of the `interrupted()` API.

Invisible token (C#/Go). In C# and Go, once a cancel is issued on a cancellation token, the status of the token cannot be reverted. Consequently, the type of mistaken clearance in Java does not exist in C# or Go. However, a cancel request may still get dropped during its propagation: since the cancellation token is typically not a global object, developers need to pass the token through function parameters to ensure the token is available through the chain of method calls. If the token is not passed to a long-running function f, cancel would be greatly delayed until the execution returns to a caller of f that has access to the token. This contributes to close to 15% of cancel-related bugs in C# and Go.

Making things more complicated, unlike Java, C# and Go allow canceling a thread through different cancellation tokens, each representing different semantics—one token

```
// Cancel recipient
class SomeTask {
  private CancellationToken systemCancelToken;

  void doWork(CancellationToken userCancelToken) {
        ...
        libraryMethod(userCancelToken); //
             systemCancelToken invisible to libraryMethod
  }
}
```

Listing 6. One type of invisible token (aspnetcore-5936)

might represent requests from end users; one might represent requests from a periodic timer; and so on. As a result, programmers may pass some tokens to a function, but forget some others, causing certain cancel requests to be dropped, as shown in Listing 6. Note that, a function typically only allows one cancellation-token parameter. Consequently, the onus is on developers to be aware of what tokens exist in the current context and when or how to combine them into one token to pass to a callee function—not a trivial task.

Lessons Learned. This type of bug can be detected by static checkers: if a function f has a cancellation-token parameter, its caller function F should pass every cancellation token *tok* visible in F to f. In fact, such a checker is included in the .NET SDK, a set of libraries that provide support for development for C#[23]. We apply this checker to the latest versions of ASP.NET Core and Roslyn, and report the results in Section 7.5.

5.3 Cancel-fulfill bugs

Once a cancel is correctly initiated and propagated to the target, the target task must process the cancel request, stopping its execution, releasing resources, and reverting or invalidating shared states so that other tasks, including a potential re-submission of the current task, can proceed correctly. This is unsurprisingly the most difficult aspect of cancel, contributing to about one third of all the bugs in our study.

5.3.1 Cancel not checked. Sometimes, a successfully delivered cancel request is not immediately checked by the target task, causing severe cancellation delays.

In Java, the complexity is that explicit cancel checking is not always needed. Once the internal cancel flag is set by the system, the target thread will throw an `InterruptedException` once it executes a blocking Java API like `sleep`, `poll`, and others. Consequently, if the target thread invokes some of these APIs from time to time, explicit checking is not needed. However, if a long-running code-region, like a loop, does not call any such APIs, explicit checks using APIs like `isInterrupted` or `interrupted` are needed. Lacking such explicit checks are the root causes behind several bugs in Java systems, like HIVE-16078 and HBASE-10575.

In C# and Go, similar problems occur if a long-running function never checks its parameter cancellation token.

Table 5. Cleanup issues breakdown

	Count
What type of cleanup defect?	
- Incorrect: wrong API or cleanup semantics	10
- Incomplete: did not clean up all data	14
- Missing: no cleanup performed	4
- Unordered: clean up data in a wrong order	3
- Other	3
Where is data requiring cleanup located?	
- Heap	27
- Persistent data	7
How should data be cleaned up?	
- Invalidate, revert or reset data	13
- Release resource (lock, thread, etc.)	13
- Delete file from disk	2
- Other	6

Lessons Learned. For C# applications, we have implemented a static checker to detect this type of bug (Section 7.4). For Go applications, implementing an accurate checker is difficult, as the `Context` variables contain many fields and could be used for many different purposes other than cancel. Automatically detecting this type of bug in Java programs is feasible. We leave this to future work.

5.3.2 Cancel not carried out. This type of bug occurs when the target task makes no attempt to stop its execution after it becomes aware of the delivered cancel request.

Our study has only seen this type of bugs in the context of the Java built-in mechanism. Specifically, an `InterruptedException` is thrown by a Java library API. This exception is caught by the caller function but the handling block is essentially empty. There are many bugs of this type (e.g., HBASE-3064, HBASE-10472, HIVE-15997, KAFKA-5833, KAFKA-1886).

Comparing with other cancel mechanisms, an `InterruptedException` contains the least semantic information—it is unclear which task initiated the cancel and for what reason. This may be why some of these catch blocks are empty.

Lessons Learned. Although the root cause here differs slightly from the "Cleared interrupt" bugs in Section 5.2.2[3], they both can be detected by a checker that searches for problematic catch blocks of `InterruptedException`, which we will discuss in Section 7.1.

5.3.3 Defective cleanups. When responding to a cancel request, a task needs to not only stop itself, but also to release resources that it acquired earlier and clean up changes it made to shared data. Doing so in a coordinated, correct, and

[3]The cancel-target task has no cancel handling across the call chain for bugs here, but has the right handling in a caller in "Cleared interrupt" bugs.

efficient way is challenging. Unsurprisingly, bugs that occur during this process are particularly common, contributing to more than 20% of all the bugs in our study.

What went wrong? There are mainly four types of mistakes in a cancel cleanup, as shown in Table 5.

First, the cancel handler changes the values of some variables in an attempt at cleanup, but the resulting values lead to failures (10 bugs in our study). For example, in SOLR-8372, upon the cancel of a *recovery* task, the *update log* this *recovery* task has been working on should remain in "inactive" state until recovery is restarted. However the cleanup logic mistakenly puts the update log into "active" state, which had the serious consequence of potential data loss. The fix was simply not to make that state change.

Next is incomplete cleanup, where the task attempted to clean up data but did not do so comprehensively (14 bugs). For example, in CASSANDRA-7803, *compaction result* files were written during the *compaction* task. The files could be written in a regular location or a temporary location, depending on the configuration. The cleanup logic removed the regular files but not the temporary ones, which could quickly fill the disk and make the application unusable.

Completely missing cleanup, where no steps are taken to clean up any data related to the task, occurred in 4 bugs. In HBASE-13877, a *TableFlushProcedure* task is canceled. However the task simply ceases execution without any additional steps taken. The data modified by the task (*Memstore Snapshot*) is not invalidated and may get reused by subsequent tasks, causing data corruption or data loss.

Finally, there are 3 bugs where the cleanup routine works on shared variables in an incorrect order, causing coordination problems with other tasks.

What data is at the center of defective cleanup? Unlike crash handling, cancel handling is carried out by the cancel target, an actively running task, and hence needs to clean up not only persistent but also heap data it has touched. In fact, for the majority of clean-up bugs (80%), heap, instead of persistent data, is the target of defective cleanup.

In our study, a canceled task T typically does not hold a close dependency with other running tasks—otherwise, T typically would not be canceled, or its dependent tasks would be canceled altogether. Consequently and fortunately, there is typically not too much heap data to clean. What needs to be cleaned are mainly low-level resources, such as locks or thread pools; or shared data structures related to system activities or persisted information. The latter includes things like task tracking, i.e. what tasks are running, have run, or about to run in the system, e.g. the `ZoneSubmissionTracker` object in Hadoop; pointers to persisted user data e.g. the `DataTracker` object in Cassandra, which maintains references to all database tables; and other system metrics or metadata, such as the `StorageMetrics` object in Cassandra which tracks disk usage, and the `RoutingNode` object in Elasticsearch, which maintains shard status information. This relatively focused target of cleanup may help future research to automate data cleanup.

Occasionally, a task which produces a large amount of intermediate results needs to be canceled. Fortunately, in most cases we have seen, the system already has a transaction-style design, where all intermediate data is buffered in a cache. The cleanup only needs to update the cache metadata correctly.

In the cases where persistent data is the target of defective cleanup, most often the data are temporary files local to a task, which are not properly deleted or invalidated. In three cases, however, the persistent data are shared by other system activities, and defects in cleaning up this data prevent the broader system from performing correctly.

What does the patch do? Most commonly, the patch releases resources, invalidates or reverts the data modified by the task. Releasing resources, such as locks, threads, and cancellation tokens, is straightforward. Often, the original task already has the correct resource release routine. However, upon a task cancel, that routine is short circuited. The patch simply makes sure the complete release routine is followed.

How to correctly invalidate or revert the data varies from case to case. Sometimes, the task needs not keep track of the modifications it has performed: for example, in CASSANDRA-5481, a task needs to reset a shared connection/cursor object on cancel, which does not require information about the history or the state of the task. But in other cases, a task must track information about modifications it has made: in CASSANDRA-15674, a task makes a single modification to *totalDiskSpaceUsed* on the shared *SystemMetrics* object, and should remember to decrement by this same value upon cancel. One challenge in performing this type of clean up is knowing, among the various heap data modified by a task, which requires cleaning and which type of cleaning.

Lessons Learned. As evidenced by the examples above, defective cleanups have severe consequences and are common. It is important to tackle these bugs.

Detecting the complete absence of cleanups is relatively easy. Whenever a cancel handler only ceases the execution and performs no cleanup, a warning should be issued. Some of these bugs can even be automatically fixed: in many cases, one just needs to re-throw the interrupt to the caller that contains the correct clean-up logic (e.g., HBASE-7711).

Some incomplete cleanups are caused by short-circuiting a correct clean-up routine. Particularly, exceptions may be thrown during the clean up, either due to unexpected task states or a system API hitting the original interrupt signal again. Incorrect handling of such a double-exception may skip the remainder of the cleanup routine, causing incomplete cleanups (HIVE-15997). Automated checkers can be developed to search for this type of bug.

Existing tools that detect resource leaks during exception handling [25] and cancellation-token leaks [21] can be applied to detect those resource leak problems.

Detecting incorrect cleanup or general missing cleanup is the most challenging and requires more research. One possible research direction is to consolidate cleanup steps to help detect and fix defective cleanups. In many bugs, the related cleanup steps were interspersed across the task. However, when they were combined or compared together, it was clear that they were not comprehensive or correct. Sometimes cleanup for one task should have been identical to another. For example, in SPARK-1396, a scheduler had two methods, *handleCancel* and *abortStage*. These should have performed the exact same cleanup steps, but for each method steps were implemented separately and non-comprehensively. The fix was to combine the cleanup logic so that it was shared. Or, the cleanup on task cancellation was very similar to the steps performed on task completion (e.g. removing a task from a registry when it is completed or canceled), and deficiencies were clear on consolidation.

Finally, given our observation that the target of cleanup is often a small set of system data structures, future research may use data-flow analysis to remind developers about what data should be cleaned, and to potentially synthesize invalidating/reverting methods for the small number of data structures that are the target of most cleanup.

5.4 Discussion: cancel mechanisms

5.4.1 Built-in mechanisms. A natural question to ask is whether different built-in cancel mechanisms cause different cancel usage issues. Some types of bugs are common no matter what mechanism is used. For example, "overlooking triggers" contribute to 19% and 26% of bugs in Java and C#/Go, respectively; "defective cleanup" contribute to 20% and 24% of bugs in Java and C#/Go, respectively.

However, there are also many types of bugs that occur particularly often in Java systems, reflecting limitations of Java's built-in cancel mechanism:

1) "Cleared interrupt" bugs (Section 5.2.2) only occur in Java programs, as neither C# nor Go allows clearing an already issued cancel request. Note that, it is natural for Java to allow clearing a cancel signal received by a thread, because each thread has only one internal cancel flag no matter how many different cancel initiators and how many different cancel contexts there might be. This limitation also influences the next two types of bugs in Java.

2) The "Late polling" bugs (Section 5.2) in theory could exist in programs written in any languages, but were only seen in Java programs by us: the use of custom cancel-flag loops is very common in Java programs and yet very rare in C#/Go programs, probably due to the limitation of Java built-in cancel mechanism as discussed above.

3) "Cancel not carried out" bugs (Section 5.3) in theory could exist in programs written in any language, but were only seen by us in Java programs. We believe this is again related to the above limitation of Java cancel mechanism. In C# and Go, a nice effect of using a `CancellationToken` as one of a task's function parameters is that it makes clear from the function protocol that the task is designed to be cancellable. The rich semantics behind cancel tokens also helps developers decide how to treat each cancel request. In contrast, in Java, `interrupt()` is available on threads by default but there is no guarantee threads respond to the interrupt, and indeed often do not.

Of course, the mechanisms in C# and Go are not perfect either. In addition to the common problems they face, such as "defective cleanup", they are particularly susceptible to "invisible token" problems (Section 5.2.2). Furthermore, the design of mixing cancel signals with other information in the `Context` variable in Go introduces challenges for both developers and researchers in designing cancel-related analysis tools.

5.4.2 Custom mechanisms. Some of the systems we studied contain components specially built to assist with cancel functionality. These components offer features that may mitigate root cause cancel issues discussed previously, and so may be of interest. We share examples of a few such constructs here.

Cancellable Task interfaces. While Java threads by default provide a method to cancel tasks, i.e. built-in `interrupt()`, a few systems provide an alternative interface to be used by cancellable tasks. At a bare minimum these interfaces declare a "cancel" method that task developers must implement, in some cases encouraging developers to sidestep built-in "interrupt" and associated problems.

For example, the *Interruptible* interface in Cassandra's "concurrent" package declares, in addition to the main task method `run()`, a method named `interrupt()` that requires implementation by developers. Though simple, this design advantageously makes explicit the task should be cancellable and actively requires cancel implementation, whereas for other task constructs, for example a generic thread, the need for cancel might not be apparent, and developers might not check for interrupts or passively ignore interrupt exceptions as we have seen. (And, an examination reveals all existing implementers of this interface do indeed handle cancel).

Some interfaces go further and include partial mechanism implementation. The abstract class *CancellableTask* in Elasticsearch's *tasks* package provides a non-overridable, pre-implemented `cancel` method which sets a member field cancel flag `isCanceled` to false (and which task execution code should check). The class also includes the status method `isCanceled()`, which may help avoid misuse problems that occur when using the built-in API to check interrupted status. We must note, however, there is a downside to side-stepping built-in interrupt entirely: if the task uses built-in blocking Java methods - e.g. sleep - it will not be able to exit these methods prematurely, as we have seen.

Interfaces may also include post-cancellation methods that developers can implement to perform cleanup or other

related tasks. *LifecycleTransaction* in Cassandra's *db* package provides, in addition to a cancel method, an `onAbort()` hook which is called after cancellation is processed. This may encourage developers to implement or consolidate cleanup logic, helping prevent missing or incorrect cleanup issues.

"Uninterruptible" interfaces. Conversely one system provides an "uncancellable" interface that allows users to run code sections without interruption: the *UninterruptibleThread* abstract class in Spark's "util" package allows users to define "uninterruptible" code sections that will complete in their entirety - if `interrupt()` is called on the thread, it will be suppressed until the uninterruptible code section completes. One area where this might be useful is for cleanup steps which must be executed in their entirety after the task is canceled: some issues we have seen arise from cleanup steps failing to complete due to interrupt during cleanup itself. An examination reveals that some implementations of this interface indeed use this functionality for cleanup. However, this design is susceptible to problems if not used carefully: if an uninterruptible code section uses an operation that blocks indefinitely, the thread may never respond to a cancellation request.

Task dependency tracking. One of the biggest categories of cancel issues is overlooking triggers, of which a common trigger is cancellation of a parent or associated task. Thus using constructs that track related or dependent tasks and help propagate cancel between them may be valuable.

For example, some systems provide a task tracking service or "task manager" that maintains a list of scheduled or running tasks, usually by requiring that all task executions be launched through the manager. The task manager may additionally be designed to track task dependencies: e.g. the *TaskManager* shared class in Elasticsearch's "tasks" package require that submitted *Tasks* contain an "id" and "parentId". All task executions are initiated through the task manager using the manager's `register` or `registerAndExecute` methods. Running tasks and their children can thus be tracked and cancellations, which must also go through the manager (via `cancelTaskAndDescendants` method), can be propagated to all dependent tasks.

6 Symptoms of Cancel-Related Bugs

Not all the bug reports specify the exact failure symptoms. We categorize the ones that describe the symptoms in Table 6. As we can see, the symptoms vary, and can be severe.

Resource leaks. Resources acquired during task execution, including locks, buffers, and others, might not be released due to defective cleanup (Section 5.3.3). Furthermore, if a cancel does not take effect, the task thread itself may be leaked, which may be especially problematic if the thread pool has a fixed size. For example in SPARK-1582, work done by a Spark *Executor* thread was no longer needed, but a cancel was delayed (sometimes indefinitely) and the thread was not made available to perform other work.

Table 6. Cancel-related bugs: symptoms

Symptom Category	Issues
Resource leaks	30
Performance issues	29
Broken task API	17
Data corruption/loss	5
Incorrect reporting	10
Unspecified	65
Total	156

Broken Task API. Unsurprisingly, incorrect cancellation might break the API used to submit or manage tasks. For example, in HDFS-12518, a critical task cannot be re-executed, due to the task not cleaning up its status when canceled. In SPARK-8132, *no* subsequent task for a multi-stage user job is able to be launched due to incorrect cleanup.

Data corruption/data loss. Many tasks might perform operations on user data, and a broken cancel can corrupt in-memory data used to service user requests, as well as cause persistent data to be lost - a very serious issue. For example, a silently dropped cancel signal in a callee led a caller to put incomplete (i.e. corrupted) in-memory values of user computations into a shared cache. Later user jobs would use these invalid values and give wrong results. (SPARK-1602).

Performance issues. While cancellation itself should generally lead to improved performance, as resources previously used by a task can be freed for other work, broken cancel handling can put the system in an unanticipated state that causes degraded performance or unresponsiveness.

In HIVE-13858 an interrupt signal was dropped, leading to an infinite loop in a task, which made access to a portion of system I/O impossible. This could cause unavailability of the entire cluster. Similarly, in CASSANDRA-11373, incomplete cleanup led to an infinite loop and CPU saturation.

In elasticsearch-75316, how frequently cancel would be used was underestimated, and inefficient cancel handling led to a 50x increase in latency for normal user requests. The patch was to make cancel handling more efficient.

Incorrect reporting to users. Lastly, mistakes in cancel functionality might lead to incorrect reports to users. For example a system might report to the user that a job has been canceled when in fact it was not (HIVE-14942, SPARK-18665, influxdb-13681). Or, conversely, the system might report that a job has not been canceled when indeed it has (SPARK-2666).

7 Task Cancel Anti-Patterns

Root causes of cancel bugs are varied and sometimes complex, but we find that a few types of bugs are associated

Table 7. Anti-pattern instances found in Java and C# applications

	HBase	Hive	Spark	Kafka	Solr	Cassandra	Hadoop	es	ASP.NET Core	Roslyn
Unhandled IE in loop (Java)	5	2	0	0	0	1	13	0	-	-
API misuse (Java)	3	2	0	7	5	0	0	0	-	-
Uncanceled child tasks (Java)	1	2	0	0	0	0	9	0	-	-
Ignored tokens (C#)*	-	-	-	-	-	-	-	-	34/112	120/179
Tokens not passed (C#)**	-	-	-	-	-	-	-	-	9	9

* Our analyzer result / CodeRush analyzer (simulated) result
** .NET analyzer (simulated) result

with clear anti-patterns that are detectable by static code analysis. This section presents our experience of designing and evaluating a few anti-pattern checkers.

We have implemented a checker for each of the anti-patterns below using CodeQL [1], a publicly available static analysis tool. CodeQL takes as input *queries* which are a set of conditions on the application source code's call graph, control flow, dataflow graph and other information (e.g. object hierarchies). Queries are language specific, so for each anti-pattern and language, we have constructed a single query that describes the anti-pattern and can be run on all applications of that language, using CodeQL's command line tool or web interface. The results of queries are references to problematic section of source code (file and line number). The queries associated with each anti-pattern can be viewed at a publicly available repository [3].

Note that, code snippets that match an anti-pattern may not all cause severe failures, but are frequently harmful to the software in the long run if not fixed. We will discuss this in detail when we comment on the severity of each anti-pattern.

Also note that, these checkers mainly tackle low hanging fruits of cancel-related bugs, with more complicated bugs waiting to be tackled by future work. We are aware of similar checkers for the two C# anti-patterns, which we will discuss in details in Section 7.4 and 7.5. There may be similar checkers for the Java anti-patterns, although we are currently not aware of them. Our main goal here is to show that it is feasible to detect cancel-related code defects through simple static checking, and that many cancel-related defects exist even in the latest versions of these popular Java and C# applications.

7.1 Unhandled Interrupt Exception (Java).

Anti-pattern. An `InterruptedException` is caught inside a loop body, but in the catch block there is no handling - no control flow to exit the loop (i.e. no `break` statement, `return` statement or rethrown exception in the AST), and the interrupt flag is not reset via `t.interrupt()` on thread `t`. In addition, we also check via dataflow analysis that the thread is indeed interrupted somewhere in the codebase.

Rationale. This anti-pattern is closely related to "cleared interrupt" bugs (Section 5.2.2) and "cancel not carried out" bugs (Section 5.3.2). Its severity has been explained in these earlier sections. Note that, in this anti-pattern, we particularly look for problems inside a loop, as it is especially problematic there: without proper cancel handling inside a loop, a task may never cease execution or incur particularly long delays (HADOOP-6221,HBASE-3064).

Severity. There is one scenario where the impact of this anti-pattern may be mitigated: the program may use a custom cancel flag together with an `interrupt` call to cancel a task. In that case, an unhandled interrupt exception may not have a big impact, as long as the remainder of the loop iteration does not take long time to execute. Having said that, this type of implementation is still problematic and makes code maintenance difficult: what if an expensive operation is added near the end of the loop iteration? What if the task initiator deems the use of flag redundant in the presence of the interrupt call and removes the former?

Results. Our checker finds 21 cases of this anti-pattern in the latest versions of 4 Java applications in our benchmark suite (Table 7). Our manual checking of these 21 cases shows that 14 of them are truly instances of this anti-pattern; 2 of them are false positives (a corner case in CodeQL control-flow analysis misses the fact that the exception handler does stop the task execution); 5 of them may be considered false positives: the exception handler sets a flag, which defers the actual handling to a later point in the loop, which may or may not cause perceivable delay in the cancel handling.

7.2 Interrupt API Misuse (Java).

Anti-pattern. A thread calls `Thread.interrupted()` inside an `InterruptedException` catch block.

Rationale. This anti-pattern is inspired by a few API-misuse bugs discussed in Section 5.2.2 (e.g., Listing 5). When an `InterruptedException` is triggered by a library method in thread `t`, the interrupt flag is almost always cleared and should be reset by invoking `t.interrupt()` if the exception is to be handled by the caller. If a `t.interrupted()` is invoked instead, this is frequently a typo, as this API is

designed to clear the interrupt flag, effectively a no-op inside the catch block. It may also be used inside a condition check, as it returns the status of the flag before clearing - `e.g. if (t.interrupted())`, - but when such checking occurs inside the catch block it is even worse, as library methods likely will have unset the flag before the check, and the logic inside the condition will never execute.

Severity. This API misuse can cause an interrupt to be dropped. Consequently, handling/cleanup logic that exists elsewhere may not be executed, causing functional problems.

Results. Our script finds 17 instances of this anti-pattern in 4 applications, as shown in Table 7. Our manual examination did not find any false positives.

7.3 Cancel not propagated to dependent tasks (Java)

Anti-pattern. A task instantiates a Java `Timer` and starts a child task (wrapped in a `TimerTask` interface) using a Java `Timer` object but does not cancel the `Timer` and `TimerTask`: either it does not maintain the reference to the `Timer` or it does not explicitly call `cancel()` on the `Timer` or `TimerTask`.

Rationale & Severity. This anti-pattern is related to some of the "Overlooking triggers" bugs discussed in Section 5.1.1. Java's built in `Timer` is one of the mechanisms used for scheduling single or periodic task executions on a separate thread. If the child task launched using the `Timer` (or `Timer` itself) is not canceled when the parent is canceled, then at a minimum, this lack of cancellation will leak resources. Note that, this anti-pattern focuses on `Timer`-based parent-child task dependency, because these type of child tasks are typically scheduled periodically and hence lead to more severe impact if not properly canceled.

Results. Our script finds 12 instances where a timer and associated tasks are started but not canceled. Three of these instances are false positives: in 2 cases, the reference to the `Timer` is embedded in a nested class, and hence is missed by our CodeQL-based static checking; in one case, the `Timer` task is only started during system shut down, and hence its leakage does not really cause problems.

7.4 Ignored cancellation tokens in loop (C#)

Anti-pattern. A method containing a loop accepts a `CancellationToken` parameter `ct`, but does not check the token via `ct.IsCancellationRequested`, `ct.CanBeCanceled` or `ct.ThrowIfCancellationRequested()`, anywhere inside a loop. Nor does it pass the token as an argument to any function calls inside the loop.

Rationale & Severity. The rationale of this anti-pattern has been discussed in Section 5.3.1. For a similar reason as discussed in Section 7.1, we focus on loops in this anti-pattern, for their bigger performance impact.

Results. Our analyzer found 154 cases of this anti-pattern (34 in ASP.NET Core and 120 in Roslyn). Manual checking finds 4 of these to be false positives: in 3 cases, a token is used via an indirect reference or reflection; in 1 case, a method that operates on a token instead of using it as a signal.

We also investigated a similar analyzer that is part of CodeRush [2], a popular debugging and code analysis extension for VisualStudio. The CodeRush analyzer warns if a token is not checked anywhere inside in a method. We have simulated the CodeRush analyzer using CodeQL and find 112 and 179 instances in ASP.NET Core and Roslyn, respectively. In one regard, our analyzer is stricter: if a token is checked somewhere in a method but not in a loop, our analyzer will flag it as a warning but the CodeRush analyzer will not. But, unlike the CodeRush analyzer, our analyzer does not check methods that do not contain loops.

7.5 Token not passed - .NET analyzer (C#)

We also applied an analyzer included as part of the .NET compiler platform (Roslyn). That Roslyn built-in analyzer checks if a `CancellationToken` is passed via parameter to a method M, but M does not pass the token to its calee C which optionally accepts a token parameter (optional arguments are a feature of the C# language). This anti-pattern is related to the "invisible token" bugs discussed in Section 5.2.2.

Simulating this anti-pattern using CodeQL, we find 9 instances each in the latest version of ASP.NET Core and Roslyn. Our manual checking finds no false positives.

7.6 Anti-pattern limitations

While these checkers have been inspired by and cover some of the bugs in our study, there are still many bugs that cannot be covered by our checkers, for various reasons. In some cases a bug manifests due to reasons logically different from those covered by our checkers: for example, a cancel is dropped due to a semantic bug in a custom mechanism, rather than API misuse or an unhandled interrupt exception.

In other cases, conditions added to our antipatterns to reduce false positives thereby introduce false negatives: for example, we search for empty interrupt exception handling specifically inside loops, but empty handling outside loops can also cause bugs.

Finally, our checkers are designed around common usage patterns and may miss other valid forms of usage: for example, we assume a cancel-supporting method is one that accepts a context or token explicitly as a top-level parameter; our checkers will ignore methods where the context or token is passed implicitly, say as a member field of another parameter.

8 Related Work

Our study is the first empirical study of task cancellation patterns and failures in concurrent systems to the best of our knowledge. Nevertheless, several related works have discussed general exception handling problems in the past.

The problem of empty exception handlers was discussed by Yuan et al. in the study of real-world failures of distributed

systems and by Fu and Ryder in the context of analyzing exception-chain of Java programs [8, 26]. Our work is orthogonal to their research, as we particularly focus on bugs related to task cancel. As discussed in Section 5, only a small portion of cancel-related bugs are due to empty exception handlers — those 6 "Cancel not carried out" bugs in Java and some of those 16 "Dropped cancel" bugs in Java. Because of the task-cancel context, why these bugs' catch blocks are empty, how to fix them, their failure symptoms, and how to generalize them into anti-patterns are all different from generic empty handler problems (e.g., the anti-pattern in Section 7.1 does not just look for empty catch blocks).

While our work discusses how cancel signals may fail to propagate to the target tasks (Section 5.2) in concurrent systems, previous work studied how incomplete error propagation could occur in file systems and storage device drivers [12, 24]. Since previous work looks at propagation through function error-code return, it is orthogonal to our study.

Past studies about general cloud system failures [11, 18] have identified error/fault handling to be a common cause, contributing to 18% of software-related failures in one study [11] and 31% of software-bug incidents in another study [18]. Both categorized error/fault handling problems into two or three major categories, including "error/fault detection", "error propagation", and "error handling". This taxonomy is similar to how we categorize cancel-related bugs at the highest level. The similarity ends here. Since both previous studies focus on general cloud failures, neither goes deep into the error/fault handling problems. The examples of detection, propagation, and handling problems there are very different from the cancel initiation, propagation, and fulfillment bugs discussed in this paper.

A Java textbook [9] has listed five possible reasons behind task cancel: (a) user-requested cancel, (b) time-limited activities, (c) application events, (d) errors, (e) shutdown. In our cancel feature study, we want to see what are the common reasons and trigger events behind task cancel in modern concurrent systems. Our study led to a categorization (Table 3) that is related but not the same as the textbook listing.

9 Future Research Directions

In this section we highlight a few potential areas for future research.

Cancellation in other languages. Different languages may have attributes which affect what types of cancel issues manifest. For example, our study focuses on garbage-collected languages; languages with manual memory management (e.g. C++) may see other cancel issues, e.g. stemming from explicit deallocation.

Cancel programming models and language features. As discussed in Section 5.4, different built-in cancel mechanisms and language constructs offer different support and challenges to developers. While we present some examples of custom cancel constructs in Section 5.4, more extensive exploration and evaluation of cancel-related designs and models are needed.

Bug-detection and other developer tools. Although this work presents static tools to detect certain classes of cancel bugs, there are still many cancel bugs that are not covered by our static checkers. More static or dynamic detection and diagnosis tools are needed.

Other kinds of developer tools may also assist in cancel implementation. For example, in Section 5.3.2 we describe how `InterruptedException` often contains the least semantic information about the source of cancel; it may be worth exploring whether developer tools, such as IDE plugins that detect and provide this contextual information, can help guide proper implementation.

10 Conclusions

Task cancellation is critical to the efficiency, availability, and operational flexibility of concurrent systems. This paper presents a comprehensive study about how task cancel is used and what type of bugs are related to task cancel in popular distributed and concurrent systems written in Java, C#, and Go. This study reveals the complexity of implementing correct and efficient task cancel, and motivates future research to offer better system support for task cancel.

11 Acknowledgements

We thank the reviewers for their insightful comments, and Mahesh Balakrishnan for shepherding this work. The authors' research is supported by NSF (grants CCF-2119184, CCF-2028427, CNS-1956180, CCF-1837120, CNS-1764039), the CERES Center for Unstoppable Computing, the Marian and Stuart Rice Research Award, and gifts from Microsoft and Facebook.

References

[1] [n.d.]. CodeQL. https://codeql.github.com

[2] [n.d.]. CRR0038 - The CancellationToken parameter is never used. https://docs.devexpress.com/CodeRushForRoslyn/119693/static-code-analysis/analyzers-library/crr0038-the-cancellation-token-parameter-is-never-used

[3] [n.d.]. Github - cancellation-study-osdi. https://github.com/whoisutsav/cancellation-study-osdi

[4] [n.d.]. Golang. https://go.dev

[5] [n.d.]. Runnable. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runnable.html

[6] Stephen Cleary. 2019. *Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming.* O'Reilly Media.

[7] Terry Crowley. 2016. How to Think About Cancellation. https://terrycrowley.medium.com/how-to-think-about-cancellation-3516fc342ae

[8] Chen Fu and Barbara G. Ryder. 2007. Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In *29th International Conference on Software Engineering (ICSE'07).* 230–239. https://doi.org/10.1109/ICSE.2007.35

[9] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. 2006. *Java concurrency in practice.* Pearson Education.

[10] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis. In *SOSP*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 66–83.

[11] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670986

[12] Haryadi S. Gunawi, Cindy Rubio-González, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast-08/eio-error-handling-occasionally-correct

[13] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *SOSP*.

[14] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: crowdsourced data race detection. In *SOSP*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 406–422.

[15] Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer. 2013. How to Cancel a Task. In *Multicore Software Engineering, Performance, and Tools*, João M. Lourenço and Eitan Farchi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–72.

[16] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *ASPLOS*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 517–530.

[17] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *SOSP*, Tim Brecht and Carey Williamson (Eds.). ACM, 162–180.

[18] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What Bugs Cause Production Cloud Incidents?. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. ACM, New York, NY, USA, 155–162. https://doi.org/10.1145/3317550.3321438

[19] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. 2021. Automatically detecting and fixing concurrency bugs in go software systems. In *ASPLOS*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.).

[20] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2014. Towards General-Purpose Resource Management in Shared Cloud Services. In *10th Workshop on Hot Topics in System Dependability (HotDep 14)*. USENIX Association, Broomfield, CO. https://www.usenix.org/conference/hotdep14/workshop-program/presentation/mace

[21] Microsoft. 2021. CA2000: Dispose objects before losing scope. https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca2000

[22] Microsoft. 2021. Cancellation in Managed Threads. https://docs.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads

[23] Microsoft. 2021. Code analysis in .NET. https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/overview

[24] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error Propagation Analysis for File Systems. *SIGPLAN Not.* 44, 6 (jun 2009), 270–280. https://doi.org/10.1145/1543135.1542506

[25] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2013. Hector: Detecting Resource-Release Omission Faults in error-handling code for systems software. In *DSN*.

[26] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[27] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *SOSP*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 116–131.

Automatic Reliability Testing for Cluster Management Controllers

Xudong Sun[†], Wenqing Luo[†], Jiawei Tyler Gu[†], Aishwarya Ganesan[‡], Ramnatthan Alagappan[‡], Michael Gasch[‡], Lalith Suresh[‡], Tianyin Xu[†]

[†]University of Illinois at Urbana-Champaign [‡]VMware

Abstract

Modern cluster managers like Borg, Omega and Kubernetes rely on the *state-reconciliation* principle to be highly resilient and extensible. In these systems, all cluster-management logic is embedded in a loosely coupled collection of microservices called *controllers*. Each controller independently observes the current cluster state and issues corrective actions to converge the cluster to a desired state. However, the complex distributed nature of the overall system makes it hard to build reliable and correct controllers – we find that controllers face myriad reliability issues that lead to severe consequences like data loss, security vulnerabilities, and resource leaks.

We present Sieve, the first automatic reliability-testing tool for cluster-management controllers. Sieve drives controllers to their potentially buggy corners by systematically and extensively perturbing the controller's view of the current cluster state in ways it is expected to tolerate. It then compares the cluster state's evolution with and without perturbations to detect safety and liveness issues. Sieve's design is powered by a fundamental opportunity in state-reconciliation systems – these systems are based on state-centric interfaces between the controllers and the cluster state; such interfaces are highly transparent and thereby enable fully-automated reliability testing. To date, Sieve has efficiently found 46 serious safety and liveness bugs (35 confirmed and 22 fixed) in ten popular controllers with a low false-positive rate of 3.5%.

1 Introduction

Modern cluster managers like Kubernetes [11], Borg [80], Twine [77], Omega [72], and vSphere [20] break down cluster-management logic into a fleet of microservices, called *controllers* [27]. For example, in Kubernetes, *all* the cluster-management logic is encoded in different controllers. Today, thousands of controllers are implemented by commercial vendors and open-source communities to extend Kubernetes with new capabilities [42, 68, 74, 78]. Controllers manage everything from application lifecycles (e.g., provisioning, upgrades, autoscaling) to stateful services, storage, networking, and integrations with cloud providers [41, 53, 57, 60, 71].

These cluster managers follow the *state-reconciliation principle* for resilience and extensibility [7, 27]. In this design, each controller continuously monitors a subset of the cluster state and reconciles the *current* state of the cluster to match a *desired* state. The cluster state is typically hosted in a logically centralized, highly available data store (e.g., etcd or ZooKeeper). In Kubernetes, entities like pods, nodes, volumes, and application instances are represented as objects in the cluster state. An auto-scaling controller might thereby monitor an application-group object for the number of currently active replicas and scale it to match the desired replica count. The design allows cluster managers to be 1) *resilient*: controllers can independently fail and pick up from where they left off, and 2) *extensible*: supporting a new feature or application is a matter of adding a custom controller that manages a set of custom objects as part of the cluster state.

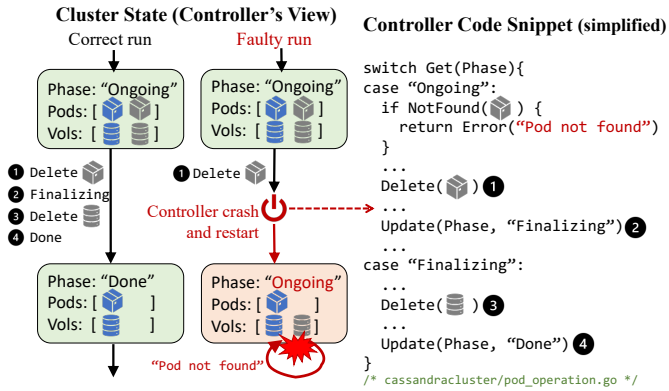


Figure 1: **A bug in a Cassandra controller detected by our tool, Sieve [30].** The controller cannot recover from an intermediate state introduced by Sieve using a crash. As a consequence, the controller cannot auto-scale the Cassandra cluster and leaks storage resources. The bug has been fixed. The code snippet is significantly simplified for clarity; the real code spans 70+ functions and 2,000+ lines of Go.

Despite the importance and prevalence of custom controllers, ensuring their reliability is challenging. Controllers run within complex, dynamic, and distributed environments. They must safely drive the system to desired states while tolerating unexpected failures, network interruptions, and asynchrony issues. If controllers are not robust to these circumstances, they lead to severe consequences such as application outages, data loss, and security issues. Buggy controllers have indeed caused many real-world problems [31, 38, 51, 52].

For example, Figure 1 shows a bug in a Kubernetes controller for managing Cassandra [30]. The bug prevents the Cassandra cluster from auto-scaling and leaks storage resources (decommissioned volumes in gray are never deleted). This is because the controller lacks crash safety – it fails to recover from an intermediate state due to a crash between deleting a Cassandra pod and updating the `Finalizing` phase.

The above crash-safety bug is only one of the myriad kinds of reliability issues that affect controllers. We find that controllers also experience bugs caused by state inconsistencies due to asynchrony effects and bugs caused by uncoordinated concurrent interactions between controllers. Existing testing techniques are either too specialized for certain types of bugs or require expert guidance in the form of formal specifications or carefully-crafted test inputs (§8). We instead seek a solution that is broadly applicable across controllers and is capable of *automatically* detecting a wide range of bugs.

Contributions. In this paper, we present Sieve, the first automatic reliability-testing technique for cluster-management controllers. Sieve drives unmodified controllers to their potentially buggy corners by systematically and extensively perturbing the controller's view of the cluster state in ways it is expected to tolerate. Sieve then compares the cluster state's evolution with and without perturbations to automatically detect safety and liveness issues.

Sieve is highly usable. It does not require 1) formal specifications of the controller or the cluster manager, 2) hypotheses about vulnerable regions in the code where bugs may lie, or 3) highly specialized test inputs. It does not rely on expert-written assertions either. Sieve requires only a manifest for building the controller image and basic test workloads. Sieve's testing is then fully automatic. This degree of usability is key to making reliability testing broadly accessible to the rapidly increasing number of custom controllers.

Sieve is powered by a fundamental opportunity in state-reconciliation systems – controllers interact with the cluster state via *state-centric interfaces*. State-centric interfaces perform semantically simple operations on the cluster state (e.g., reads and writes) and deliver notifications about cluster-state changes; the objects that flow through the interfaces typically have a uniform schema. Therefore, state-centric interfaces are highly introspectable and hence an ideal vantage point to observe and perturb a controller's view of the cluster state.

Sieve leverages the fact that a controller's actions are strictly a function of its view of the current cluster state. We thus test a controller by exhaustively introducing state perturbations through failures, delays, and reconfigurations. These are circumstances that reliable controllers are expected to tolerate. Currently, Sieve supports three typical perturbation patterns that expose controllers to 1) intermediate states (Figure 1), 2) stale states (or past cluster states), and 3) unobserved states due to missing some cluster state transitions (§3.1).

For each pattern, Sieve automatically generates test plans that cover all possible perturbations during an execution of the controller under test. Test-plan generation is based on analyzing a controller's behavior and the cluster-state evolution during reference executions. Sieve effectively avoids redundant and futile test plans to maximize test efficiency.

Sieve automatically detects buggy controller behavior using differential test oracles that compare the cluster-state transitions with and without perturbations. This comparison is feasible because a controller's behavior is reflected in the sequence of cluster-state transitions. The differential oracles are often more effective than searching for errors in logs and more comprehensive than human-written assertions (§3.6).

Key results. We implemented Sieve for Kubernetes controllers. Sieve requires only a manifest for building the controller image and basic workloads (e.g., a scale-up-and-down workload for an autoscaling feature). Sieve's testing is then fully automatic. We evaluated Sieve on ten popular open-source controllers of various kinds, from either commercial vendors or official projects. Sieve found 46 new bugs in total, among which 35 have been confirmed (22 fixed) after we reported them. Notably, these are deep semantic bugs that Sieve detected without any expert guidance. The bugs have severe consequences, including application outages, security vulnerabilities, resource leaks, and data loss. Sieve is highly efficient—all controllers could be tested in under seven hours on a cluster of 11 machines, representing a typical nightly test. Sieve also has a very low false-positive rate of 3.5%, making its testing results trustworthy.

Summary. The paper makes four main contributions:

- We present the first automatic reliability-testing technique for state-reconciliation systems: exhaustively perturbing the controller's view of cluster states and using differential oracles on the cluster state evolution to detect bugs.
- We design and implement Sieve, a system that uses our proposed technique to automatically test *unmodified* cluster-management controllers in Kubernetes.
- Sieve has already improved reliability for ten popular open-source controllers by virtue of bugs it found that were then fixed by developers. It is practical to run Sieve regularly.
- We have made Sieve publicly available at `https://github.com/sieve-project/sieve`, with instructions to reproduce all discovered bugs.

2 Background and Motivation

Modern cluster management and control plane designs follow the state-reconciliation pattern, where control loops reconcile the current state of the cluster to conform to a desired state. Kubernetes, like its predecessors Borg and Omega, follows the idea of reconciliation control loops for resiliency [27]. Similarly, vSphere [81] and NSX [82] continuously monitor and correct deviations from declaratively-specified desired

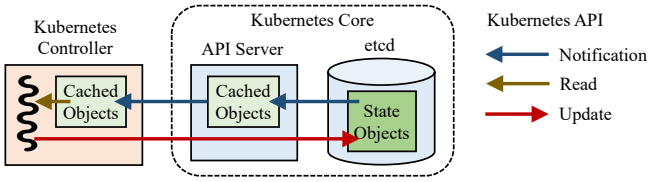


Figure 2: **Interactions between a controller and other Kubernetes components and the state-centric APIs.**

states to manage virtual machines and networks. These systems rely on a clean separation between the cluster state and the control plane logic [27]; the state is represented as mere data (e.g., JSON objects), and the control plane logic queries and manipulates the state programmatically.

We now give a brief overview of state reconciliation and cluster-management controllers. We also present the urgent need for automated reliability testing and our insights.

State reconciliation by example. We use Kubernetes as a representative example to present the basics of state reconciliation. Figure 2 illustrates Kubernetes' architecture. Kubernetes' core comprises an ensemble of *API servers* and a highly available, strongly consistent data store (etcd) that houses the *cluster state*. The cluster state is represented by a collection of *objects*. Every entity in the cluster has a corresponding object in the cluster state, including pods, volumes, nodes, and groups of applications. All other components in Kubernetes interact with the cluster state via API servers.

All cluster-management logic is encoded in *controllers* that are clients of the API servers. The controllers continuously monitor a part of the cluster state and perform state reconciliation whenever the current state does not match the desired state. The controllers perform reconciliation by querying and manipulating the state objects via a client library that exposes a state-centric interface. This interface provides notifications, reads, and writes involving the cluster state objects.

This design enables Kubernetes to be highly extensible: supporting a new application or feature is a matter of adding a new controller and a corresponding set of custom object types to the cluster state; it does not require changes to the client library or interface. The design also allows controllers to be loosely coupled, which improves resilience: controllers can independently fail and new controller instances can resume reconciliation without fail-over logic.

Figure 3 shows how a collection of controllers coordinate in a loosely coupled manner. To deploy a ZooKeeper cluster running on Kubernetes, the user creates a ZooKeeper object which specifies the desired state of the ZooKeeper cluster (e.g., replica count, version, storage size) via the Kubernetes command-line tool. The ZooKeeper controller receives a notification that a ZooKeeper object was created. To drive the system to the desired state, it updates the cluster state by creating a StatefulSet object (an abstraction to run stateful applications). Then, a StatefulSet controller is subsequently

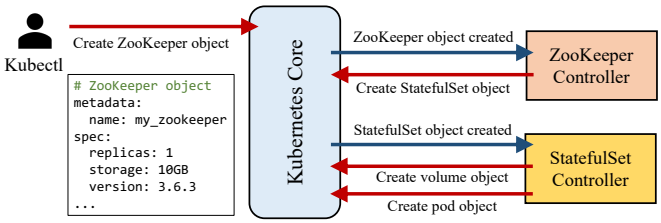


Figure 3: **The workflow of deploying ZooKeeper on Kubernetes using a ZooKeeper controller.**

notified about the StatefulSet object being created, which in turn creates pod and volume objects to run the containerized ZooKeeper nodes. While not shown in the figure for brevity, this subsequently leads to more controllers like a scheduler, a storage controller, and worker nodes being activated to bring up the actual containers and volumes. Similarly, if the user then edits the desired state of the ZooKeeper object (e.g., the number of replicas), it triggers a similar sequence of reconciliations by different controllers, as each tries to make minor adjustments to get to its appropriate desired state.

The need for automated reliability testing. Kubernetes' extensibility has led to a thriving ecosystem with *thousands* of domain-specific controllers implemented by commercial vendors and open-source communities [41, 42, 53, 60, 68, 71, 74]. For example, OpenShift, an enterprise Kubernetes platform from Red Hat, provides 130+ custom controllers that extend Kubernetes [17]. All these controllers represent critical infrastructure, making their correctness paramount. As shown by many real-world problems [31, 38, 51, 52] and our evaluation results, designing and implementing reliable controllers is challenging – many popular, mature controllers misbehave under faults, delays, and asynchrony with severe consequences.

However, controller reliability is notoriously hard to ensure. A developer faces the fundamental challenge of 1) anticipating all possible *views* of the cluster state at the controller (compounded by asynchrony) and 2) safely reconciling to the required desired states from any of these points. We observe that manually-written test suites do not sufficiently test a controller's reliability (§6).

Unfortunately, existing testing techniques are either too specialized for certain bug types (e.g., crash-recovery bugs or concurrency bugs) and cannot address the broad range of controller bugs; or require expert guidance in terms of formal specifications of the system, crafted heuristics, or hypotheses on vulnerable code regions (§8). We seek a solution that is easy to use and broadly applicable to unmodified controllers.

Our insight. To overcome the above challenges, we 1) automatically and extensively perturb an unmodified controller's view of the cluster states in ways it is expected to tolerate, and 2) automatically flag safety and liveness issues using differential oracles that compare the evolution of cluster states with and without perturbations. This degree of automation

and unintrusiveness is enabled by the fundamental nature of state-reconciliation systems. That is, these systems often have a simple and highly introspectable state-centric interface with which controllers interact with the cluster state. Such interfaces essentially do no more than reads and writes, or receive notifications regarding state-object changes. All objects share a common schema, which makes any arbitrary object highly introspectable. For example, all objects in Kubernetes have an identical set of fields representing their metadata. This enables a degree of automation that is hard to achieve otherwise.

3 Sieve Design

Sieve is an automatic reliability testing tool for cluster management controllers. It checks whether the controllers under test can correctly operate the system under common perturbations (due to unexpected faults and inherent asynchrony) and detects bugs that lead to safety and liveness issues at the development time. Sieve is automatic—it tests unmodified controllers and does not rely on formal specifications or controller-specific assertions. Sieve is effective—it focuses on well-defined, highly-targeted perturbations that reliable implementations are required to tolerate.

Sieve perturbs the controller's view of the cluster state based on three broad patterns that expose the controller to 1) intermediate states, 2) stale states, and 3) unobserved states. We discuss the three patterns and their rationales in §3.1. Note that these are not the only patterns in which faults can occur, but cover a broad range of faults that a component in a distributed system is expected to handle gracefully. Sieve can be extended to incorporate other patterns in the future.

Sieve tests controllers with the following workflow:

- *Collecting reference traces* (§3.2). Sieve starts by learning how a controller behaves in the absence of faults (under test workloads) and records the state transitions in reference traces. To do so, it instruments the state-centric interfaces used by the controller to interact with the cluster state.
- *Generating test plans* (§3.3). Sieve then analyzes the reference traces to generate *test plans*. A test plan describes a concrete perturbation. The test plan specifies *what* faults to inject and *when* to inject them to effectively drive the controller to see a target cluster state.
- *Avoiding ineffective test plans* (§3.4). To achieve high test efficiency, Sieve prunes redundant or futile test plans. For example, it avoids a test plan if it is clear that it cannot causally lead to a target cluster state.
- *Executing test plans* (§3.5). Sieve executes each test plan using a test coordinator. The test coordinator monitors the cluster-state transitions during testing and injects the specified faults according to the test plan's specification.
- *Checking test results* (§3.6). Sieve has generic, effective, differential oracles to automatically check test results. The oracles detect buggy controller behavior by comparing the cluster-state evolution between the reference and test runs.

Sieve deals with non-deterministic elements of the cluster state during testing to minimize their impact on test plan generation and test oracles (§3.7). Specifically, Sieve identifies non-deterministic state objects and fields and excludes them.

Usage. To use Sieve, one needs to provide two inputs: 1) a manifest that specifies how to build and deploy the controller under test, and 2) a set of test workloads that exercise end-to-end behavior of the controller under test. The two inputs are mostly available in mature controller projects, as they are needed for controller development and deployment. In our experience, finding them is straightforward.

3.1 Perturbing A Controller's View of The State

Sieve operates under the assumption that a controller follows the state-reconciliation principle, which receives a sequence of *notifications* about the changes to the cluster states and outputs a corresponding sequence of *updates* to the cluster states. Sieve aims to affect the outputs of a controller by perturbing its view of the cluster state. These perturbations are produced by injecting targeted faults (e.g., crashes, delays, and connection changes) when specific cluster-state changes (*triggering conditions*) happen.

Notably, the perturbation strategy allows Sieve to *decouple policy from mechanism*. The decoupling makes it easy to extend existing policies or add new policies by orchestrating the underlying perturbation mechanisms. Specifically, a policy defines a view Sieve exposes to the controller at a particular condition, while the mechanism specifies how to inject faults to create the view. Sieve automatically generates test plans for each policy; each test plan introduces a concrete perturbation based on a specification of a triggering condition and a fault to inject when that condition happens.

Sieve currently supports three patterns to perturb a controller's view. Crucially, these perturbations drive a controller to states that it is expected to tolerate. They represent valid inconsistencies in the view that a controller could see due to common faults as well as the inherent asynchrony of the overall distributed system. Over time, we hope to add more perturbation patterns.

Intermediate states. Intermediate states occur when controllers fail in the middle of a reconciliation before finishing all the state updates they would have otherwise issued. After recovery (e.g., Kubernetes automatically starts a new instance of a crashed controller), the controller needs to resume reconciliation from the intermediate state left behind.

Figure 4 illustrates how Sieve tests the official RabbitMQ controller with intermediate-state perturbations and reveals a new bug. The test workload attempts to resize the storage volume from 10GB to 15GB. The resizing is implemented with two updates: 1) updating `VolCur` to 15GB; 2) updating

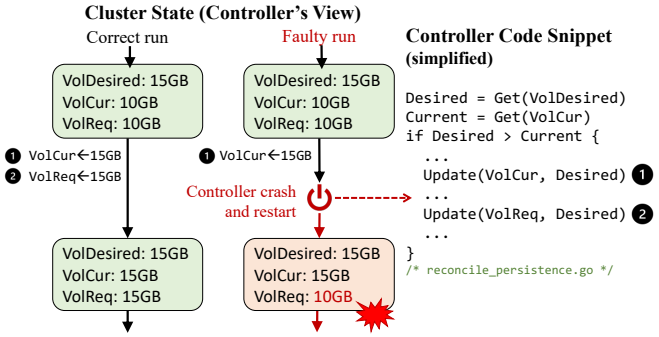


Figure 4: **An intermediate-state bug in a RabbitMQ controller detected by Sieve [70].** The controller fails to recover from the intermediate state introduced by Sieve; the controller does not successfully resize the storage volume.

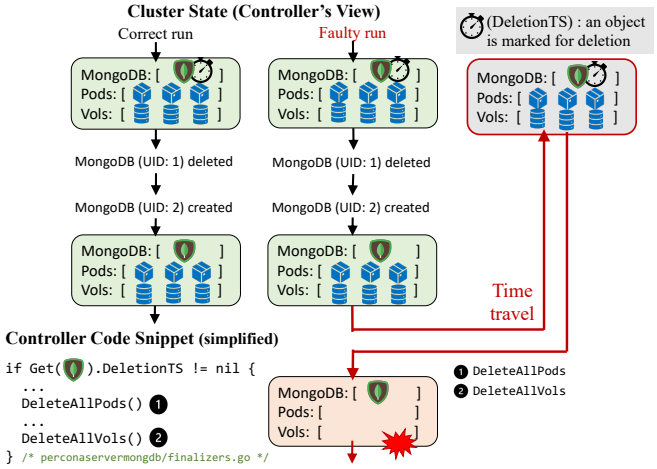


Figure 5: **A stale-state bug in a MongoDB controller detected by Sieve [43].** The controller experiences a "time-travel" and observes a stale state. It makes wrong reconciliation action based on the stale state (deleting all the pods and volumes) which leads to application outages and data loss.

`VolReq` to 15GB which triggers Kubernetes to resize the volume. The controller issues updates when `VolCur` is smaller than the desired volume size. During testing, Sieve crashes the controller between the two updates, which creates an intermediate state where `VolCur` is updated, but `VolReq` is not. The controller cannot recover from the intermediate state and the resizing never succeeds. The bug has been fixed with 700+ lines of Go code to revamp the volume resizing logic. In addition, the developers added eight new tests along with the fix to exercise how the controller handles different intermediate states, which is what Sieve performs automatically.

Stale states. Controllers often operate on stale states, due to asynchrony and the extensive uses of caches for performance and scalability [26]. As shown in Figure 2, controllers do not directly interact with the strongly consistent data stores, but are connected with API servers. The states cached at API servers could be stale due to delayed notifications. Controllers are expected to tolerate stale views that lag behind the latest states maintained in the data store.

Tolerating stale views correctly is nontrivial. For example, a Kubernetes controller's view may "time travel" to a state it observed in the past. Time traveling occurs when there are multiple API servers operating in a high-availability setup, when the controller reconnects to a stale API server that has not yet seen some updates to the cluster state. The reconnection can be triggered by failover, load balancing, or reconfigurations. Controllers are expected to recognize the stale state [18], instead of treating it as a new, unseen state.

Figure 5 illustrates how Sieve tests Percona's MongoDB controller with stale-state perturbation and reveals a new bug that leads to both application outages and data loss. To support graceful MongoDB cluster shutdowns, the controller waits to see a non-nil deletion timestamp (`DeletionTS`) field attached to the state object representing the MongoDB cluster (a common practice to give systems time to react to an impending deletion [23]). When the controller sees this change, it deletes all the pods and volumes of the MongoDB cluster.

Sieve drives the controller to mistakenly delete a live MongoDB cluster by introducing a time-travel perturbation. With a workload that first shuts down a MongoDB cluster and then recreates a new instance of the same cluster, Sieve waits till the cluster is recreated and then introduces a time-travel perturbation. The perturbation causes the controller to see the deletion timestamp being applied to the *already-deleted cluster*. Consequently, the controller mistakenly shuts down the newly created cluster. This revealed that the controller should be checking for the UIDs of clusters, not just their names.

Unobserved states. By design, controllers may not observe every cluster-state change in the system. The full history of changes made to the cluster state is prohibitively expensive to maintain and expose to clients [76]. Controllers are hence expected to be designed as *level-triggered* systems (opposed to being *edge-triggered*), i.e., a controller's decision must be based on the currently observable cluster state (level) [21], not on seeing every single change to the cluster state (edge).

Figure 6 illustrates how Sieve tests Instaclustr's Cassandra controller using unobserved-state perturbations and reveals a new bug that leads to resource leaks and service failures. The test workload first scales down and then scales up storage volumes of the Cassandra cluster. During scale-down, the controller removes volumes when it learns that the corresponding pods were marked for deletion (a non-nil deletion timestamp field is set on the pod object, similar to the previous example). The pods' lifecycles (including deletions) are managed by a built-in controller called a StatefulSet controller. Sieve pauses notifications to the Cassandra controller for a window such that it does not see these deletion marking events by the StatefulSet controller. This causes the Cassandra controller to not delete the corresponding volumes even though it has the

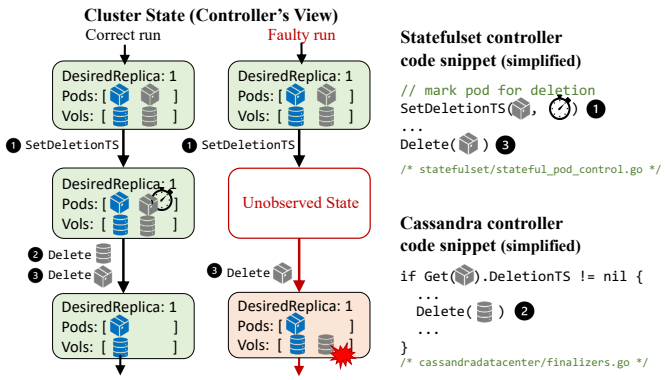


Figure 6: **An unobserved-state bug in a Cassandra controller detected by Sieve [29].** The controller misses a transient state where the pod has a non-nil deletion timestamp. It thus fails to delete the volumes, leaking storage resources. The bug also prevents new Cassandra pods from rejoining.

right information to make that call (i.e., its view has volumes created by it that do not have pods attached to them).

Hence, the volume never gets deleted, leaking the storage resource. The bug also prevents the controller from scaling the Cassandra cluster – newly-created pods try to reuse the dangling volumes and cannot rejoin using the cluster metadata already in them (as it represents a node that was decommissioned). The bug has been fixed by adding a pre-deletion hook – a coordination mechanism in Kubernetes that allows the Cassandra controller to complete the required cleanup operations before the pods can be deleted [9].

3.2 Collecting Reference Traces

Sieve starts by learning how a controller behaves in the absence of faults. To do so, Sieve interposes around the state-centric interfaces used by the controllers to interact with the cluster state. All modern cluster managers have unified, well-defined client libraries based on state-centric interfaces. Taking Kubernetes as an example, any interaction with the cluster state (exposed by the API servers) goes through a small, well-defined set of client APIs that read, modify, or receive notifications about state objects. They are used by every controller that interacts with the Kubernetes API servers. To support Kubernetes controllers, Sieve decorates 10 functions in the client library and this interposition is fully automated (§4).

With the interposition in place, Sieve learns every cluster-state change notification that the controller receives, as well as any reads and writes attempted by the controller to the cluster state or to the local cache of the cluster state maintained by the client. Sieve then runs each test workload supplied by the developer and collects the following two reference traces:

- *Controller trace.* A series of events observed via the interposition of client APIs, including notifications about state changes, entry and exits of each reconciliation cycle, and client-API invocation by the controller and their arguments.
- *Cluster state trace.* The initial cluster state and the sequence of state changes (object creations, modifications, and deletions), collected using public APIs of the cluster manager.

The controller trace is used for generating test plans (§3.3) and the cluster-state trace is used by test oracles (§3.6).

3.3 Test Plan Generation

Sieve generates a set of test plans for each test workload for which it has collected reference traces. Each test plan specifies a perturbation to inject during the workload.

A test plan is represented by a self-contained file that describes a test workload, a list of faults to inject during the workload run, and the triggering condition for when to inject each fault. Sieve currently supports several primitives that test plans can compose to introduce complex faults: 1) crash/restart a controller, 2) disconnect/reconnect a controller to an API server, 3) block/unblock a controller from processing events, and 4) block/unblock an API server from processing events. When an executed test plan reveals a bug, the test-plan file is sufficient to reproduce the bug.

Figure 7 shows a simplified test-plan file generated by Sieve. Each element in `faults` specifies the fault to inject (`faultType`) and the triggering conditions (`triggers`). Each element in `triggers` specifies a triggering condition, that causes the specified fault to be injected before or after a particular cluster state change if executed. A composite triggering condition can be specified in `compositeTrigger` by combining multiple conditions in `triggers` with boolean operators. For example, `t1 & t2` means the fault is only injected when both `t1` and `t2` are triggered. In Figure 7, `trigger1` is the only required condition to inject the fault. Similarly, composite faults can be constructed (e.g. crashing a controller after `t1` and restarting it after `t2`).

We now present the basic rules Sieve applies to compute test plans that exercise one of the three patterns in §3.1. We later describe how Sieve avoids ineffective test plans in §3.4. Optionally, one can customize patterns by implementing new rules or manually writing test plans.

Intermediate-state rule. For a controller, Sieve generates test plans that force all possible intermediate states and exposes them to the controller. To do so, Sieve analyzes the reference controller trace and marks the sequence of state updates made by the controller within each reconciliation loop. Concretely, for every reconciliation that issues multiple state updates, $U_1, U_2, ..., U_n$, Sieve generates one test plan per state update U_i, where Sieve crashes the controller after it issues U_i. When the controller restarts after the crash, it is presented with the target intermediate state.

Stale-state rule. For stale states, Sieve generates test plans that make the controller travel back in time and see stale states that it has already observed. Concretely, Sieve checks the controller trace for a notification-update pair (N, U), such that observing N results in an update U (see §3.4.1). It then

```
testWorkload: resizePVC
faults:
- faultType: crashController
  triggers:
  - triggerName: trigger1
    triggerAt: afterControllerIssues
    stateChange:
      beforeChange: 'VolCur:10GB'
      afterChange: 'VolCur:15GB'
  compositeTrigger: trigger1
```

Figure 7: **A test plan generated by Sieve.** This is a simplified view of the test-plan file that detected the bug in Figure 4. This test plan crashes the RabbitMQ controller right after the controller updates `VolCur` from 10GB to 15GB. Sieve learns every state change issued by the controller via the state-centric interfaces (e.g., `Update` in Table 1).

searches for a subsequent state-change notification N' which has a conflicting effect with U (e.g., U deletes an object and N' creates the same object). With time traveling, if the controller mistakenly issues U after seeing the stale state N, it could corrupt the newer cluster state as notified by N'.

Sieve generates test plans that 1) block a reserved API server to prevent it from advancing its own state after it sees N, 2) after the controller sees N', time-travel the controller to see N by reconnecting the controller to the reserved stale API server, and 3) unblock the stale API server; so, the introduced staleness is only transient—both the API server and the controller catch up eventually. We focus on deletions for U because they are destructive operations.

Unobserved-state rule. For unobserved states, Sieve generates plans that skip states that a controller might observe during normal executions, but could potentially miss in the presence of faults. Sieve checks the controller trace to find pairs of notifications (N, N') in which N' is the closest subsequent notification that cancels the effect of N. Sieve generates test plans that 1) block the controller to prevent it from seeing N, and 2) unblock the controller when N' arrives. Such a test plan causes the controller to miss cluster states from N (inclusive) up to N' (exclusive).

3.4 Avoiding Ineffective Test Plans

Sieve may potentially generate a large number of test plans using rules specified in §3.3. For example, in stale-state testing, Sieve might identify every notification the controller receives as a point to inject staleness, therefore generating test plans for every received notification. For example, the naïve rule above for stale states would generate 140,000+ test plans for the MongoDB controller in Figure 5. It is therefore key to prune ineffective test plans.

As a guiding principle, we prune a test plan if the test plan does not introduce an intermediate-, a stale- or an unobserved-state that can affect the controller's outputs, or the introduced state is identical to states introduced by other test plans. This naturally requires Sieve to have a clear notion of what input events affect the controller's outputs.

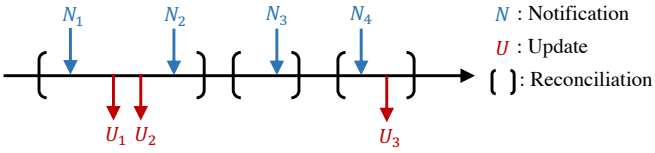


Figure 8: **Causality rules used by Sieve.** For simplicity, in this figure the object pertaining to each notification is immediately read by the controller. (N_1, U_1), (N_1, U_2), (N_3, U_3) and (N_4, U_3) are causally related according to the rules in §3.4.1. N_2 is *not* causally related to any update, given the earliest-reconciliation rule.

3.4.1 Pruning by Causality

If a controller makes an update U based on a notification N, we consider N and U to be causally related. We consider a pair (N, U) that is not causally-related to be irrelevant from a testing standpoint, because perturbing N will not affect U.

Inferring causality between events is generally a challenge in distributed systems. By focusing on the "narrow waist" of state-centric input and output events of the controller under test, we are able to design simple yet effective rules for Sieve to infer whether a pair (N, U) is causally related. These rules are lenient and only introduce false positives at best, but not false negatives. False positives increase testing times by generating redundant test plans, whereas false negatives risk reducing test coverage that could miss bugs. While causal tracing support [58] for Kubernetes is currently in its infancy [6], we might be able to leverage it in the future.

Sieve currently considers a pair (N, U) to be causally related if both the following conditions holds (Figure 8 exemplifies the two causality rules):

- *Read-before-update rule:* the object pertaining to N is read by the controller before it issues U;
- *Earliest-reconciliation rule:* N and U happen in the same or adjacent reconciliation cycles. The rationale is that controllers always issue updates relevant to N in the earliest possible reconciliation cycle after N is received.

For stale- and unobserved-state testing, Sieve only generates test plans involving a N if it has at least one causally-related U. We find pruning test plans by causality effective, especially when there are many notifications due to other activities irrelevant to the controllers under test.

3.4.2 Pruning Unsuccessful Updates

Sieve ignores any update U that does not change the current cluster state. Sieve checks whether an update U is successful based on whether U triggers a state change ΔS of the cluster state. This information is typically encoded in the return value of the U operation.

For stale-state testing, Sieve further ignores an update U that, if issued again, does not change any of the subsequent cluster states, (i.e., there does not exist an N' that is affected by U). Sieve checks whether the state objects updated by U would later be conflicted with N'.

The rationale for the above pruning is straightforward. If an update does not change the current cluster state, it is unlikely to cause new states in the execution. If an update U cannot affect any future cluster states, it would not perturb the controller's execution under the time-travel pattern either, i.e., if U is issued again, it would not corrupt any future state.

In practice, we found that many controllers issue unsuccessful updates that do not actually change the cluster state, including pathologically frequent ones caused by inefficient but benign behavior (see Section 5.2).

3.5 Test Plan Execution

Every test plan is executed by the Sieve test coordinator running in the testing cluster. The coordinator faithfully executes the test plan by running the test workload and injecting the faults specified in the test plan. Specifically, the test coordinator monitors state transitions of both the controller's view of the cluster state as well as the cluster state as seen by API servers. This is done based on the interposition described in §3.2; it allows the test coordinator to intercept and take actions (e.g., injecting faults) when state transitions happen. If the observed state transition matches the triggering condition ΔS specified in the test plan, the coordinator marks the condition as matched. The coordinator injects a fault (e.g., a controller crash) once all the corresponding conditions are matched. Most of the interposition and injection are done through the client APIs. But, for stale-state testing, the coordinator also needs to interpose at the API server (to make an API server stale).

As a concrete example, to execute the test plan in Figure 7, the test coordinator monitors every state transition issued by the RabbitMQ controller. The coordinator marks `trigger1` in the test plan as matched when it observes a state transition that updates `VolCur` from 10GB to 15GB. Since `trigger1` is the only required condition in the test plan, the coordinator injects a controller crash right after `trigger1` is matched. If the test plan specifies multiple faults, the coordinator injects them one by one according to the specified order.

The test coordinator also records the cluster states in a trace during testing, which will be compared with the reference cluster-state trace (§3.2) to detect buggy behavior.

3.6 Differential Test Oracles

Sieve has generic, effective oracles to automatically detect safety and liveness issues. The oracles detect buggy controller behavior based on the *cluster states* during and at the end of the test run. The goal is to validate that the testing traces are free of safety and liveness issues, in addition to monitoring anomalous controller behavior (e.g., crashes and hangs). Developers can also add domain-specific oracles.

In our experience, many buggy controller behaviors do not show immediate or obvious symptoms (e.g., crashes, hangs, and error messages). Instead, they lead to data loss, security issues, resource leaks, and unexpected application behavior which is hard to check with oracles typically used by prior art [55, 75, 84, 88, 89]; in our evaluation, only five (out of 46) bugs can be flagged by checking for exceptions or crashes.

We therefore develop differential test oracles that compare cluster states in a reference run versus those in test runs—with inconsistencies typically indicating buggy behavior. This methodology means we need to exclude nondeterministic states and state objects affected by the perturbation (§3.7).

We found that Sieve's differential oracles vastly outperform developer-written assertions in the test suites of the controllers we evaluated, because Sieve's oracles systematically examine all the state objects and their evolution during testing. It is challenging for developers to manually codify oracles that comprehensively consider the large number of relevant states.

Note that Sieve does implement regular error checks for obvious anomalies, including exceptions, error codes and timeouts. Sieve scans the controller's log and checks whether the controller encountered any unexpected exception (i.e., `panic` in Go). Sieve also checks whether the operations in the test workload return error codes or fail to complete on time.

3.6.1 Checking End States

Sieve systematically checks the end state after running a workload. Specifically, our oracles check the count of state objects by type and the field values of all the objects after accounting for nondeterminism (§3.7). It compares the end state of the test run versus the reference run. Sieve fails the test if it finds inconsistencies between the end states and prints human-readable messages to pinpoint inconsistencies.

Such checking is effective compared to the simpler assertions that we found in test suites for the controller projects we studied. For example, in an intermediate-state bug [46], the MongoDB controller fails to create an SSL certificate used for securing communications inside the MongoDB cluster. This causes the controller to fall back to insecure communications. Such security issues do not manifest in the form of crashes or error messages. Sieve however automatically catches the bug, because the certificate object in the faulty run does not exist in the cluster state, which is different from a normal run. The bug was detected by Sieve and confirmed by the developers.

We found that none of the 71 test cases shipped with the controller has an assertion that checks the certificate object, despite the fact that enabling TLS is recommended and is the default configuration [66]. We would not be able to repurpose the assertions in these test cases to catch this bug.

3.6.2 Checking State-Update Summaries

Besides the end state, Sieve also checks how the controller updates the cluster state over time. It does so by comparing summaries of constructive and destructive state updates for each object (e.g., `CREATE` and `DELETE` operations). Such checks are complementary to the end-state checks, because a correct end state does not imply that the controller behavior is always correct during the test. We find that buggy behavior can end in correct states (same as in the reference runs).

For example, in a stale-state bug [47], the XtraDB controller mistakenly deletes the front-end proxy (which routes user requests to the XtraDB cluster), causing service unavailability. After the staleness ends, the API server and the controller eventually catch up with updated states and recreate the proxy. In this case, the end state of the proxy in the test run is the same as in the reference, but the update that deleted the proxy in the test run is buggy. Sieve detects the bug by noting that the proxy pod receives 2 `CREATE` and 1 `DELETE` operations in the faulty run, but only 1 `CREATE` in the reference run.

In another intermediate-state bug [62], the NiFi controller fails to reload configuration files. The end state is the same as a normal run; however, in the faulty run, the controller did not restart the NiFi pod to reload the configuration. Sieve flags this by noting the NiFi pod receives a `CREATE` and a `DELETE` operation (to reload the configuration) in the normal run, but neither appears in the faulty run.

Note, comparing the *sequence* of state-update is unreliable and would lead to false alarms—the sequences are not strictly the same due to concurrent controller operations. The summaries instead are robust to different event orderings.

3.7 Dealing with Nondeterminism

The shape of a state object (the set of fields and their values) might be nondeterministic. This nondeterminism affects Sieve's test plan generation and the differential test oracles. We now describe how Sieve combats this problem.

All objects have identifying metadata (e.g., a type, namespace, and name). This is key for Sieve to identify two instances of the same object, both within a run (e.g., checking for conflicting operations in the stale-state rule) and across runs (e.g., comparing configurations of objects across runs).

Sieve identifies nondeterministic field values by running the test workloads without perturbation multiple times when generating reference runs, and then comparing the values of each field in each state object.

Objects whose identifying metadata is nondeterministic are excluded from test plan generation and subsequent steps, because Sieve cannot reliably match them across runs or setup triggering conditions for them. If other kinds of fields have nondeterministic values (typically IP addresses, timestamps, or even random port numbers), Sieve does not exclude the object but simply masks the field values. Note that these two rules still allow Sieve to spot unexpected changes to the set of fields on the object (e.g., missing deletion timestamp fields).

API	Component	Instrumentation
`reconcileHandler`	Client	Log entry and exit
`Get`, `List`	Client	Send objects to coordinator
`Create`, `Update`, `Patch`	Client	Send objects to coordinator
`Delete`, `DeleteAllOf`	Client	Send objects to coordinator
`HandleDeltas`	Client	Send objects to coordinator
`processEvent`	API server	Send objects to coordinator
`Get`, `List`	Client	Add delay
`processEvent`	API server	Add delay

Table 1: **Instrumentation performed by Sieve to monitor and perturb states.** The instrumentation is automated.

In addition, Sieve provides an API for Sieve users to exclude specific state objects or fields from test plans or oracles based on domain knowledge, if needed.

4 Implementation

We implement Sieve for Kubernetes controllers. Sieve uses `kind` [10] to run a Kubernetes cluster on a single machine, so every test plan can be run entirely on one machine. Sieve configures two API servers for stale-state testing. Sieve is implemented in 5,500 lines of Python code (for test plan generation and oracles) and 3,100 lines of Go code (for automated instrumentation and fault injection).

Sieve instruments 10 API methods, representing the state-centric interface, for monitoring and perturbing states (Table 1). Those methods are in Kubernetes client libraries [13, 14] and the API server. Sieve implements an automated procedure to instrument the 10 methods using `dst` [8] to work with different versions of Kubernetes client libraries and API servers. Sieve analyzes the syntax tree for each method to insert monitoring and fault-injection code. Sieve applies the instrumentation when building the controller image. Sieve does not need to analyze or instrument the controller code.

In Kubernetes, level-triggered controllers do not immediately read notifications when they arrive [21]. Instead, the controller first updates a locally-cached view of the state objects; the controller reads from this cache when it uses `Get` or `List` APIs to query the cluster state. In causality analysis (§3.4.1) Sieve needs to know whether a notification is read before an update. To do so, Sieve analyzes the state objects updated by each notification and those read by each `Get`/`List`.

Some controllers are multi-threaded, where each thread calls a different reconcile function. Sieve uses the instrumented client libraries to obtain stacktraces whenever the controller reads or updates the cluster state (e.g., `Get`, `Create`). These stacktraces are used to differentiate between controller threads when generating and executing test plans.

5 Evaluation

Sieve's premise is that automatic and effective reliability testing for unmodified controllers is viable, by a) exhaustively

Controller	Systems	Dev.	#Stars	#Commits	#WL
cass-operator	Cassandra	DataStax	287	477	2
cassandra-operator	Cassandra	Instaclustr	227	337	2
casskop	Cassandra	Orange	177	1643	3
elastic-operator	Elasticsearch	Official	1832	3375	2
mongodb-operator	MongoDB	Percona	142	1407	5
nifikop	NiFi	Orange	101	232	3
rabbitmq-operator	RabbitMQ	Official	343	1679	3
xtradb-operator	XtraDB	Percona	302	1693	5
yugabyte-operator	YugabyteDB	Official	41	36	4
zookeeper-operator	ZooKeeper	Pravega	242	220	2

Table 2: **Kubernetes controllers used in our evaluation.** "#WL" stands for the number of different test workloads.

perturbing a controller's view of the cluster states and b) using differential oracles to flag safety and liveness issues.

We validate this hypothesis with three evaluation questions: 1) Can Sieve find new bugs in real-world controllers? 2) Does Sieve do so efficiently? 3) Are Sieve's testing results trustworthy? We answer these questions in the affirmative:

- §5.1: Sieve finds *new* bugs in *all* ten evaluated controllers, resulting in a total of 46 *new* bugs, which represent a swathe of safety and liveness issues. So far, 35 of them have been confirmed and 22 have been fixed by the developers.
- §5.2: All controllers can be tested in seven hours on a cluster of 11 machines, representing a typical nightly test. This is attributed to the effective reduction techniques which reduce test plans by 46.7%–99.6% across the controllers.
- §5.3: Sieve poses a low false positive rate of 3.5%.

Tested controllers We evaluated Sieve on ten popular controllers from the Kubernetes ecosystem for managing widely-used cloud systems (Table 2). The controllers are either developed by the official development team of the corresponding system, or by companies that have production-grade offerings around said systems. The term *operator* [12] in the project names refers to the Kubernetes design pattern of using a custom controller to manage an application.

Sieve employs 2–5 basic test workloads for each controller (Table 2). Each workload exercises a feature of the controller. Every evaluated controller supports software deployment and autoscaling, and therefore has at least two workloads. Sieve also employs workloads for controllers that support more features, such as sharding, storage resizing, reconfiguration, and load balancing. A test workload is typically implemented in 6–12 lines of code and takes 4–12 minutes to run.

It took us on average three hours to apply Sieve to each controller, which was mostly spent on understanding how to build the controller. We expect controller developers to expend much less effort to integrate Sieve in their workflow.

5.1 Finding New Bugs

Sieve finds a total of 46 *new* bugs in the evaluated controllers (Table 3). Those bugs include 11 intermediate-state bugs, 19 stale-state bugs, 7 unobserved-state bugs, and 9 bugs indirectly detected by Sieve during testing. Sieve finds new bugs in *all* the evaluated controllers. We have reported all these bugs. So far, 35 of them have been confirmed and 22 have been fixed. No bug report was rejected.

Controller	Interm. State	Stale State	Unobser. State	Indirect	Total
cass-operator	2	1	0	0	3
cassandra-operator	0	2	1	2	5
casskop	1	2	1	0	4
elastic-operator	0	2	0	0	2
mongodb-operator	2	3	1	3	9
nifikop	2	0	0	1	3
rabbitmq-operator	1	2	1	0	4
xtradb-operator	3	3	1	0	7
yugabyte-operator	0	2	1	2	5
zookeeper-operator	0	2	1	1	4
Total	11	19	7	9	46

Table 3: **New bugs detected by Sieve in each controller.**

Sieve can consistently reproduce all the 37 intermediate-, stale-, and unobserved-state bugs—running the test plan always reproduces the buggy behavior. In our experience, Sieve's reproducibility is invaluable for debugging test failures. It helps developers localize bugs in the source code and continuously iterate on bug fixes (§6).

Table 4 shows the consequences of the 46 controller bugs and an exemplar bug for each kind of consequence. We see that many bugs have severe consequences, such as application outages, security issues, service failures, and data loss. Note that these controllers are all mature projects (Table 2 and §6), suggesting that controller reliability is challenging to achieve.

The bugs that Sieve finds are deep and highly unlikely to be detected by manual testing or imprecise techniques like chaos testing or randomized fault injection tools [1, 2, 4, 5]. For example, a bug [63] in nifikop is triggered only when the controller crashes between issuing two specific state updates within one reconciliation loop (the time window between the two updates is about 0.7 milliseconds). In contrast, the test workload used for detecting the bug takes about 440 seconds to finish, and causes 481 reconciliation loops and 1,687 state updates issued by the controller. Sieve is able to detect and consistently reproduce the bug because it relies on injecting a fault precisely when a specific cluster-state change happens.

5.1.1 New Bugs Detected by Sieve

Intermediate-state bugs. Sieve found 11 intermediate-state bugs. Sieve stresses a common pattern among controllers, where they issue multiple updates per reconciliation, after the controller checks for a certain condition to hold in the cluster state. However, Sieve finds bugs when these condition checks only detect states from running the reconciliation loop in its entirety; that is, when the checks do not account for intermediate states that may arise due to controller crashes. For

Consequence	Example	# Bugs
Application outage	rabbitmq-operator-648: The RabbitMQ cluster is mistakenly turned down [69].	12
Service failure	K8SPSMDB-433: Sharding service for the MongoDB cluster wrongly terminated [44].	5
Data loss	K8SSAND-559: Storage volumes of Cassandra replicas wrongly deleted [49].	8
Reduced reliability	zookeeper-operator-314: The ZooKeeper cluster scaled down unexpectedly [90].	7
Misconfiguration	nifikop-49: The NiFi pod is not updated with new configuration [62].	6
Security issue	K8SPXC-896: TLS is not enabled for the XtraDB cluster [48].	6
Resource leak	cassandra-operator-398: Volumes used by deleted replicas never recycled [29].	7
Controller malfunc.	casskop-370: The controller stops serving scaling requests [30].	8

Table 4: **Consequences of the bugs found by Sieve (Table 3).** One bug can lead to multiple consequences.

example, in the intermediate-state bug in Figure 4, rabbitmq-operator compares `VolCur` and `VolDesired` to check whether the volume has been expanded already. However, this check assumes that all subsequent steps in the reconciliation succeed whenever this condition is satisfied. In the bug in Figure 1, the condition check cannot differentiate an intermediate state versus an unexpected faulty state. Part of the challenge is that controllers lack mechanisms analogous to write-ahead logging or journaling to guarantee atomicity of each reconcile action to enforce crash consistency. Controllers typically run as Kubernetes pods themselves and the newly created controller pod instance lacks any memory of its past execution (as they *should* – controllers must only depend on the *current state*). Sieve exposes those bugs without the need to understand source code—it systematically tests a controller with all possible intermediate states and checks for correctness.

Stale-state bugs. Sieve found 19 stale-state bugs. In our experience, it is notoriously challenging to anticipate all possible stale states. That said, we found controllers were not adequately using Kubernetes' mechanisms to tolerate asynchrony and staleness: like object versioning and unique IDs (instead of referring to objects by names, that need not be unique), or using coordination mechanisms to enforce ordering between events. Controllers also have the option to avoid staleness by using quorum reads to API servers, but this creates a scalability bottleneck as it drives more load to etcd – developers therefore choose to synchronize selectively. In general, we do not believe there is a shortcut to reasoning about any given update under all possible staleness or time-travel scenarios. Sieve therefore aids developers by systematically testing controllers under all possible time-traveling scenarios.

Unobserved-state bugs. Sieve found 7 unobserved-state bugs. We find that all of them are rooted in latent edge-triggering behavior in the controllers, that go against the Kubernetes philosophy of designing controllers to be level-triggered (§3.1). That is, these bugs arise when the controller's correctness relies on observing a specific state transition (edges), as exemplified by Figure 6. By identifying states that would be later overwritten and preventing those states from being observed by the controller, Sieve is effective at exposing unobserved-state bugs in controllers.

Bugs indirectly detected by Sieve. Sieve also finds 9 bugs that were not directly triggered by input states Sieve generated but *were still correctly flagged* by its differential oracles. All these bugs could (and do) happen in reference runs as well; but because Sieve executes many test plans, some test traces inevitably *differ* from the reference traces due to these bugs, allowing Sieve to detect them. These bugs are caused by a range of issues, including 1) controllers making incorrect assumptions about the Kubernetes API (e.g., assuming a list of pods from a query have stable ordinals); 2) spurious, dangling object creations, masked by Kubernetes' garbage collection (e.g., accidentally creating ZooKeeper pods after deleting the high-level ZooKeeper cluster object); 3) the applications managed by the controller being buggy and failing. Sieve can be extended with new perturbation patterns to systematically force out some of those bugs. For example, after understanding the root causes, we were able to reproduce two of these bugs consistently with manually written test plans.

5.1.2 Oracle Effectiveness

Sieve's differential oracles are crucial to detect buggy executions. Of the 46 newly found bugs, 45 were flagged by the differential test oracles. Checking logs for errors only flagged 5 bugs of which 4 were also found by our differential oracles.

Our end-state checker (§3.6.1) finds 28 bugs by comparing the end states of a test workload with and without perturbation. The state-update summaries checker (§3.6.2) finds 17 more bugs by checking the number of object updates through an execution. These oracles allow Sieve to detect bugs such as security and reliability issues (see §3.6) that do not manifest as simple failure symptoms (e.g. exceptions or process crashes).

The only bug that the differential oracles fail to find but is found by a regular error check in log files, is a null-pointer dereference bug [45] that causes an unexpected controller crash. Since Kubernetes automatically restarts the controller, it does not affect the end states or the state updates.

5.2 Test Efficiency

Table 5 shows the total time Sieve takes to test each controller in terms of machine hours. All experiments were run on 11 Amazon EC2 virtual machines, each with 8-core Intel(R) Xeon(R) Platinum 8259CL CPU with 2.50GHz and 32 GB memory, running Ubuntu 20.04.2 LTS.

Sieve's total testing time varies from 11.07 to 67.24 machine hours across the controllers. Sieve runs tests in parallel because every test plan is independent. With eleven virtual

Controller	Testing Time (Machine Hours)			# Test Plans
	Generation	Execution	Total	
cass-operator	0.60	43.67	44.27	218
cassandra-operator	0.49	10.72	11.21	81
casskop	0.57	12.40	12.97	125
elastic-operator	0.43	30.10	30.53	245
mongodb-operator	1.00	66.24	67.24	584
nifikop	1.17	41.61	42.78	239
rabbitmq-operator	0.47	10.60	11.07	133
xtradb-operator	1.40	62.96	64.36	395
yugabyte-operator	0.67	17.38	18.05	196
zookeeper-operator	0.33	13.75	14.08	164

Table 5: **Sieve's total testing time for each controller.**

machines, the total testing time for each controller is no more than 7 hours. Therefore, it is practical to run Sieve as a regular nightly test.

Over 95% of the testing time is spent on executing test plans. With the perturbations introduced by Sieve, a workload takes 8.8% longer to run on average. The overhead mainly comes from delays injected by Sieve for stale- and unobserved-state testing. In a few cases, when Sieve triggers bugs that lead to liveness issues, the controller hangs and triggers a timeout (by default, 10 minutes).

Sieve also spends 0.33–1.40 hours to 1) collect the reference trace and 2) generate test plans for each controller. The collected trace for each workload contains 3,386 events of notifications, updates, or reads on average. Generating test plans takes only 20 seconds for each workload on average.

Test reduction. Sieve's techniques to avoid ineffective test plans are key for tractability. Figure 9 breaks down the cumulative contribution of each technique. The baseline represents the basic rules described in §3.1 without any of the pruning techniques in §3.4. Overall, Sieve prunes away 46.7%–99.6% possible test plans across the evaluated controllers.

Specifically, pruning by causality (§3.4.1) reduces test plans by up to 95.0% across the controllers. This reduction is particularly effective for controllers that receive many notifications that are not causally related to any update. For example, mongodb-operator receives 700+ notifications regarding 20+ state objects, which are not causally related to most of its updates. This allows Sieve to prune *136,000*+ causally *unrelated* pairs of notifications and updates.

Pruning unsuccessful updates (§3.4.2) further prunes up to 75.8% of test plans across the controllers. In casskop, 60.0+% of updates issued by the controller do not affect the cluster state because the controller redundantly recreates two `service` objects that already exist. As none of these updates are relevant, Sieve excludes them when generating test plans.

Sieve finally prunes up to 72.9% of test plans across all controllers by focusing on deterministic triggering conditions (§3.7). This makes Sieve robust to many peculiar behaviors. For example, zookeeper-operator has an inefficient but benign behavior – it regularly clears the `NodePort` field of a `service`

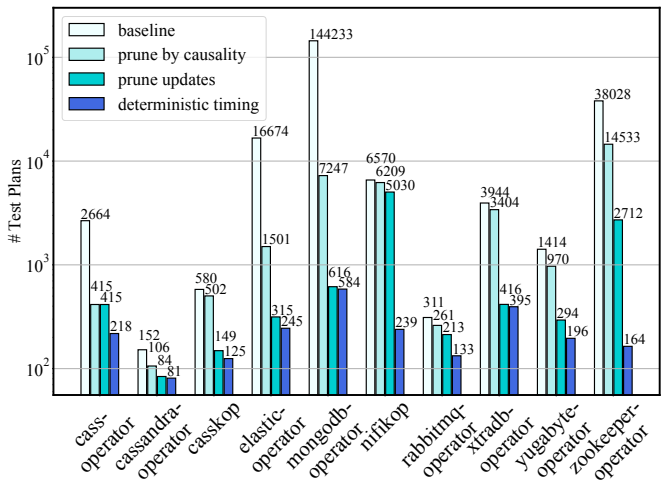


Figure 9: **Effectiveness of Sieve's test plan reduction techniques (§3.4).** The number of generated test plans is reduced by 46.7%–99.6% compared with the baseline.

object in every reconciliation, forcing Kubernetes to randomly allocate a port. This leads to thousands of state transitions with random port numbers. Sieve identifies these transitions to be nondeterministic and avoids related test plans.

5.3 False Positives

Sieve has a low false positive rate of 3.5%. It reports a total of 227 test failures for the ten evaluated controllers. 219 of them were true alarms—the test failures are caused by the 46 bugs described in §5.1 (one bug might fail multiple tests). The other eight test failures are false alarms.

The eight false alarms come from test results of three controllers (casskop, nifikop and xtradb-operator). All of them are caused by benign state transitions introduced in the faulty runs that did not happen in the reference runs.

The false alarms do not lead to opaque test failures—Sieve pinpoints the inconsistent fields. In all eight cases, the false alarms are easy to identify based on the identified fields and we could validate them by running the vanilla workload.

6 Discussion

In this section, we reflect on our experience building Sieve and studying the root causes of bugs it found (§5.1).

We find that all the evaluated controllers adopt mature software testing practices and have numerous unit, integration, and end-to-end test cases. Some even test scenarios involving faults. However, it is prohibitively difficult for developers to anticipate all possible cluster states that may occur, let alone codify them into test cases. Sieve fills this gap by exhaustively testing input states according to patterns of interest. For two bugs, Sieve detects that the initial bug fixes are deficient in covering all the conditions. *We run Sieve on the patched controllers and Sieve still detects the bugs!*

We also find that it is challenging for developers to comprehensively check test results, given the enormous state objects and their fields. Developers typically check a few fields of interest but such assertions can easily miss subtle, but serious issues (e.g., security vulnerabilities as discussed in §3.6).

We also observe that certain bugs are likely rooted in misunderstandings of Kubernetes' design and API semantics. For example, some unobserved-state bugs are caused by incorrectly assuming that every state change can be observed by the controller; some stale-state bugs can be prevented by using Kubernetes' mechanisms like resource versioning and precondition checking. We expect such problems to be more prevalent as engineers implement more and more custom controllers for their cluster management needs.

While cluster managers may avoid some classes of bugs, they come with hard tradeoffs. For example, not caching state objects at the controllers and API servers (Figure 2) could avoid stale-state bugs. However, it would introduce significant performance overheads to the controllers (memory accesses become network round trips) and make the data store a scalability bottleneck [26]. Also, transactions are not a solution for intermediate-state bugs – it would complicate the state-centric interface and prevent controllers from independently making progress regardless of failures, a key factor for resilience.

Since there is no silver bullet to implementing reliable controllers, we believe that automatic tools like Sieve are critical to cluster management reliability.

7 Limitations

Like other testing tools, Sieve is neither sound nor complete. Sieve uses specific perturbation patterns and exhaustively drives controllers to input states according to those patterns.

Sieve's differential oracles can yield both false negatives and positives. Sieve only applies its oracles on cluster states exposed by the state-centric interface. It is possible that certain application-specific states cannot be observed by the interface, which would lead to false negatives. In addition, Sieve reports false positives if the inconsistencies captured by the differential oracles are caused by benign state transitions that did not happen in the reference runs (§5.3). We found the false positive rate low (3.5%) in our evaluation.

The way Sieve deals with nondeterminism also leads to false negatives. Sieve excludes objects with nondeterministic metadata and masks nondeterministic field values in test plan generation and the differential test oracles (§3.7). This approach effectively avoids many irreproducible test plans and false positives, but also misses bugs that are triggered by states involving nondeterministic fields.

Lastly, Sieve depends on test workloads provided by the user for coverage. Implementing a test workload only takes 6-12 lines of code from our experience, but it requires domain knowledge about the controller and the system.

8 Related Work

Testing control-plane software. Modern SDNs have state-reconciliation based elements [15, 16, 19, 73] that could be tested using Sieve's methodology. A body of orthogonal work tests [28, 83] or verifies [37] how an SDN controller affects a network topology. For example, NICE [28] focuses on the boundary where controllers process packet-in/out events and uses that vantage point to automatically test for bugs. To the best of our knowledge, Sieve is the first work to focus on automatic reliability testing for cluster-manager controllers.

Testing distributed systems. Fault-injection tools [3, 25, 32, 34, 35, 55, 59, 79] have been developed for distributed systems, including chaos testing tools from the industry for cluster managers [1, 2, 4, 5]. Sieve's goals differ from those in the fault injection literature. Sieve seeks to expose controllers to as many input states as possible to test their reliability. For us, faults just happen to be a good mechanism to drive controllers to the required states. Compared to randomized chaos testing approaches that are unaware of cluster state transitions, Sieve can precisely force specific bug-triggering state transitions and consistently reproduce bugs. Furthermore, unlike prior art [32, 55, 87], Sieve is not based on an expert's hypotheses about vulnerable regions in the code under test.

A few prior tools can, in principle, expose some bugs found by Sieve. For example, concurrency-testing tools [61, 64, 65, 88] may expose bugs triggered by unobserved states (which in essence occur due to reordering of events). Similarly, tools that check for crash safety [22, 32, 55, 67] could expose bugs caused by the intermediate states. Finally, tools that inject network partitions [3], with expert guidance, could find some bugs caused by the stale-state pattern, i.e., a partition might force a controller to talk to a lagging API server (after talking to an up-to-date one). In contrast to these tools, Sieve does not target one class of bugs. Through exhaustive state perturbations, Sieve finds many kinds of bugs, essentially combining the power of prior targeted tools. Further, the chances that prior tools will find the bugs Sieve does are small, as they lack the context required to efficiently drive controllers to their buggy corners (e.g., a network-partition injector is unlikely to reliably orchestrate time-travel bugs).

Model checking. Sieve bears similarities to implementation-level model checking [40, 50, 54, 56, 61, 85, 86], in that we drive an unmodified implementation to a range of states to find bugs. Unlike model checking, Sieve does not seek to exhaustively cover the controller's state space. It instead executes developer-supplied test cases and exhaustively perturbs these test cases according to some fault patterns. Additionally, model checkers typically rely on a specification for correct behavior. While Sieve intentionally does not require hand-crafted specifications, it leans on reference traces as a partial specification of expected correct behavior.

Automation and ease-of-use. Sieve treats automation and ease-of-use as first-class design goals. Our automation is pri-

marily enabled by state-centric interfaces, which Sieve uses to produce perturbed states. Prior work has leveraged similar interface boundaries (e.g., the system-call interface [36,67]), enabling a general scheme to test multiple applications. For ease of use, Sieve does not require formal specifications [28,33,39], special test input [3, 24], code modifications [34, 61, 65], or whitebox analysis [54]. It also uses differential oracles to avoid the additional effort to supply domain-specific oracles [3, 22, 39]. However, Sieve's efficacy can be further improved with such expert guidance.

9 Conclusion

We present Sieve, the first automatic reliability testing technique for cluster management controllers. We find that Sieve is effective and practical. Sieve's usability and reproducibility play a critical role in understanding, debugging, and fixing reliability bugs. Sieve's testing technique is general and easy to extend – it separates the policy (how to perturb a controller's view of state) from mechanisms (how to realize perturbations). Hence, we are able to use the technique to detect a wide range of bugs without brittle heuristics, specifications or hypotheses. Our goal is to make Sieve a part-and-parcel of every controller developers' toolkit, and to harden the growing number of controllers that power today's data centers. We have made Sieve publicly available at https://github.com/sieve-project/sieve.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Ranjita Bhagwan, for their insightful comments. We thank Marcos Aguilera, Sujata Banerjee, Mihai Budiu, Jon Howell, Rob Johnson, Matthew Lentz, Darko Marinov, Davanum Srinivas, Chaitanya Bhandari, Yinfang Chen, Lilia Tang, and Shuai Wang for valuable feedback and discussions that helped shape this work. We thank all the controller developers who engaged with us and reviewed our reports and patches. This work was funded in part by NSF SHF-1816615, CNS-2130560, CNS-2145295, and a VMware Research Gift.

References

[1] Chaos mesh — a solution for system resiliency on kubernetes. https://dzone.com/articles/chaos-mesh-a-chaos-engineering-solution-for-system, 2020.

[2] Chaoskube: chaoskube periodically kills random pods in your kubernetes cluster. https://github.com/linki/chaoskube, 2020.

[3] Jepsen. https://jepsen.io/, 2020.

[4] Kubemonkey: An implementation of netflix's chaos monkey for kubernetes clusters. https://github.com/asobti/kube-monkey, 2020.

[5] Pumba: Chaos testing, network emulation, and stress testing tool for containers. https://github.com/alexei-led/pumba, 2020.

[6] API Server Tracing. https://github.com/kubernetes/kubernetes/pull/94942, 2021.

[7] Controllers and Reconciliation. https://cluster-api.sigs.k8s.io/developer/providers/implementers-guide/controllers_and_reconciliation.html, 2021.

[8] Decorated Syntax Tree. https://github.com/dave/dst, 2021.

[9] How finalizers work. https://kubernetes.io/docs/concepts/overview/working-with-objects/finalizers/#how-finalizers-work, 2021.

[10] kind: Kubernetes IN Docker - local clusters for testing Kubernetes. https://kind.sigs.k8s.io/, 2021.

[11] Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/, 2021.

[12] Kubernetes Operators. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, 2021.

[13] kubernetes-sigs/controller-runtime. https://github.com/kubernetes-sigs/controller-runtime, 2021.

[14] kubernetes/client-go. https://github.com/kubernetes/client-go, 2021.

[15] Open Virtual Networking. https://github.com/ovn-org/ovn, 2021.

[16] Open vSwitch: Production Quality, Multilayer Open Virtual Switch. https://www.openvswitch.org/, 2021.

[17] OpenShift: The developer and operations friendly Kubernetes distro. https://github.com/openshift, 2021.

[18] Resource versions. https://kubernetes.io/docs/reference/using-api/api-concepts/#resource-versions, 2021.

[19] VMware NSX-T. https://docs.vmware.com/en/VMware-NSX-T-Data-Center/index.html, 2021.

[20] vSphere: Unified Management for Containers and VMs. https://www.vmware.com/products/vsphere.html, 2021.

[21] What is a Level Based API. https://book-v1.book.kubebuilder.io/basics/what_is_a_controller.html, 2021.

[22] Alagappan, R., Ganesan, A., Patel, Y., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[23] Alpar, A. Using Finalizers to Control Deletion. https://kubernetes.io/blog/2021/05/14/using-finalizers-to-control-deletion/, May 2021.

[24] Alquraan, A., Takruri, H., Alfatafta, M., and Al-Kiswany, S. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[25] BASIRI, A., BEHNAM, N., DE ROOIJ, R., HOCHSTEIN, L., KOSEWSKI, L., REYNOLDS, J., AND ROSENTHAL, C. Chaos Engineering. *IEEE Software 33*, 3 (Mar. 2016), 35–41.

[26] BROOKER, M. The Fundamental Mechanism of Scaling. http://brooker.co.za/blog/2021/01/22/cloud-scale.html, 2020.

[27] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, Omega, and Kubernetes. *Communications of the ACM 59*, 5 (May 2016), 50–57.

[28] CANINI, M., VENZANO, D., PEREŠÍNI, P., KOSTIĆ, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)* (Apr. 2012).

[29] CASSANDRA-OPERATOR-398. [BUG] Reconcilation fails to delete PVCs if missing a deletion timestamp of the Cassandra pod. https://github.com/instaclustr/cassandra-operator/issues/398, 2021.

[30] CASSKOP-370. [BUG] Casskop fails to clean up PVCs and refuses to handle user requests after crash and restart. https://github.com/Orange-OpenSource/casskop/issues/370, 2021.

[31] CHEKRYGIN, I. Keep the Space Shuttle Flying: Writing Robust Operators. In *KubeCon Europe* (May 2019).

[32] CHEN, H., DOU, W., WANG, D., AND QIN, F. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering (ASE'20)* (Sept. 2020).

[33] DELIGIANNIS, P., MCCUTCHEN, M., THOMSON, P., CHEN, S., DONALDSON, A. F., ERICKSON, J., HUANG, C., LAL, A., MUDDULURU, R., QADEER, S., AND SCHULTE, W. Uncovering Bugs in Distributed Storage Systems during Testing (not in Production!). In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)* (Feb. 2016).

[34] DRĂGOI, C., ENEA, C., OZKAN, B. K., MAJUMDAR, R., AND NIKSIC, F. Testing Consensus Implementations Using Communication Closure. In *Proceedings of 2020 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'20)* (Nov. 2020).

[35] GANESAN, A., ALAGAPPAN, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)* (Feb. 2017).

[36] GONG, S., ALTINBÜKEN, D., FONSECA, P., AND MANIATIS, P. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-Thread Communication Analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)* (Oct. 2021).

[37] GUHA, A., REITBLATT, M., AND FOSTER, N. Machine-Verified Network Controllers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)* (June 2013).

[38] GUILLOUX, S. Writing a Kubernetes Operator: the Hard Parts. In *KubeCon North America* (Nov. 2019).

[39] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. Fate and Destini: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (Mar. 2011).

[40] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (Oct. 2011).

[41] HAASE, S. How an Operator Becomes the Hero of the Edge. In *OperatorCon* (May 2019).

[42] HALL, C. AWS, Google, Microsoft, Red Hat's New Registry to Act as Clearing House for Kubernetes Operators. https://www.datacenterknowledge.com/open-source/aws-google-microsoft-red-hats-new-registry-act-clearing-house-kubernetes-operators, Mar. 2019.

[43] K8SPSMDB-430. [BUG] Stale deletion timestamps lead to undesired statefulset and PVC deletion. https://jira.percona.com/browse/K8SPSMDB-430, 2021.

[44] K8SPSMDB-433. [BUG] Sharding stateful set gets mistakenly deleted when reading stale field values. https://jira.percona.com/browse/K8SPSMDB-433, 2021.

[45] K8SPSMDB-434. [BUG] Nil pointer dereference when reconfiguring spec.sharding.enabled. https://jira.percona.com/browse/K8SPSMDB-434, 2021.

[46] K8SPSMDB-578. [BUG] Failure of creating SSL-internal certificates when the controller crashes and restarts at some particular point. https://jira.percona.com/browse/K8SPSMDB-578, 2021.

[47] K8SPXC-725. [BUG] HAproxy stateful set and services get mistakenly deleted when reading stale spec.haproxy.enabled. https://jira.percona.com/browse/K8SPXC-725, 2021.

[48] K8SPXC-896. [BUG] The controller fails to set up SSL-internal certificates if a crash happens at some particular point. https://jira.percona.com/browse/K8SPXC-896, 2021.

[49] K8SSAND-559. [BUG] PVCs can be deleted mistakenly when reading stale deletion timestamp information. https://k8ssandra.atlassian.net/browse/K8SSAND-559, 2021.

[50] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI'07)* (Apr. 2007).

[51] KUMAR, H., AND ŠAFRÁNEK, J. Storage on Kubernetes - Learning From Failures. In *KubeCon North America* (Nov. 2019).

[52] LAGRESLE, M. Moving to Kubernetes: the Bad and the Ugly. In *ContainerDays* (June 2019).

[53] LANDER, R. Kubernetes Operators: Should You Use Them? https://tanzu.vmware.com/developer/blog/kubernetes-operators-should-you-use-them/, July 2021.

[54] LEESATAPORNWONGSA, T., HAO, M., JOSHI, P., LUKMAN, J. F., AND GUNAWI, H. S. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[55] LU, J., LIU, C., LI, L., FENG, X., TAN, F., YANG, J., AND YOU, L. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'19)* (Oct. 2019).

[56] LUKMAN, J. F., KE, H., STUARDO, C. A., SUMINTO, R. O., KURNIAWAN, D. H., SIMON, D., PRIAMBADA, S., TIAN, C., YE, F., LEESATAPORNWONGSA, T., GUPTA, A., LU, S., AND GUNAWI, H. S. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys'19)* (Mar. 2019).

[57] MACCÁRTHAIGH, C. PID loops and the art of keeping systems stable. https://www.infoq.com/presentations/pid-loops/, June 2019.

[58] MACE, J., ROELKE, R., AND FONSECA, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[59] MAJUMDAR, R., AND NIKSIC, F. Why is Random Testing Effective for Partition Tolerance Bugs? In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[60] MUSAJI, M. Why Operators are essential for Kubernetes. https://www.redhat.com/en/blog/why-operators-are-essential-kubernetes, Apr. 2021.

[61] MUSUVATHI, M., QADEER, S., AND BALL, T. CHESS: A Systematic Testing Tool for Concurrent Software. Tech. Rep. MSR-TR-2007-149, November 2007.

[62] NIFIKOP-49. [BUG] NiFi configuration cannot be reloaded if the controller crashes and restarts in the middle of a reconciliation. https://github.com/konpyutaika/nifikop/issues/49, 2021.

[63] NIFIKOP-79. [BUG] Nifikop fails to scale down NiFi cluster due to a crash in the middle of reconcileNifiPod. https://github.com/konpyutaika/nifikop/issues/79, 2022.

[64] OZKAN, B. K., MAJUMDAR, R., NIKSIC, F., BEFROUEI, M. T., AND WEISSENBACHER, G. Randomized Testing of Distributed Systems with Probabilistic Guarantees. In *Proceedings of 2018 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'18)* (Nov. 2018).

[65] OZKAN, B. K., MAJUMDAR, R., AND ORAEE, S. Trace Aware Random Testing for Distributed Systems. In *Proceedings of 2019 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'19)* (Oct. 2019).

[66] PERCONA DISTRIBUTION FOR MONGODB OPERATOR DOCUMENTATION. Transport Layer Security (TLS). https://www.percona.com/doc/kubernetes-operator-for-psmongodb/TLS.html, 2021.

[67] PILLAI, T. S., CHIDAMBARAM, V., ALAGAPPAN, R., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[68] PIPES, J., HAUSENBLAS, M., AND TABER, N. Introducing the AWS Controllers for Kubernetes (ACK). https://aws.amazon.com/cn/blogs/containers/aws-controllers-for-kubernetes-ack/, Aug. 2020.

[69] RABBITMQ-OPERATOR-648. [BUG] Reading stale RabbitMQ cluster information leads to unexpected StatefulSet deletion. https://github.com/rabbitmq/cluster-operator/issues/648, 2021.

[70] RABBITMQ-OPERATOR-782. [BUG] PVC expansion fails if the controller crashes in the middle of a reconciliation. https://github.com/rabbitmq/cluster-operator/issues/782, 2021.

[71] RATIS, P. Lessons Learned using the Operator Pattern to build a Kubernetes Platform. In *USENIX SREcon* (Oct. 2021).

[72] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)* (Apr. 2013).

[73] SCOTT, C., WUNDSAM, A., RAGHAVAN, B., PANDA, A., OR, A., LAI, J., HUANG, E., LIU, Z., EL-HASSANY, A., WHITLOCK, S., ACHARYA, H., ZARIFIS, K., AND SHENKER, S. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM'14)* (Aug. 2014).

[74] SOSA, C., AND BHATIA, P. Application management made easier with Kubernetes Operators on GCP Marketplace. https://cloud.google.com/blog/products/containers-kubernetes/application-management-made-easier-with-kubernete-operators-on-gcp-marketplace, May 2019.

[75] SUN, X., CHENG, R., CHEN, J., ANG, E., LEGUNSEN, O., AND XU, T. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[76] SUN, X., SURESH, L., GANESAN, A., ALAGAPPAN, R., GASCH, M., TANG, L., AND XU, T. Reasoning about Modern Datacenter Infrastructures Using Partial Histories. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'21)* (June 2021).

[77] TANG, C., YU, K., VEERARAGHAVAN, K., KALDOR, J., MICHELSON, S., KOOBURAT, T., ANBUDURAI, A., CLARK, M., GOGIA, K., CHENG, L., CHRISTENSEN, B., GARTRELL, A., KHUTORNENKO, M., KULKARNI, S., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., VENKATESAN, V., AND ZHANG, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[78] TANG, Z., LI, X., AND GUO, F. Demystifying Kubernetes as a service – How Alibaba cloud manages 10,000s of Kubernetes clusters. https://www.cncf.io/blog/2019/12/12/demystifying-kubernetes-as-a-service-how-does-alibaba-cloud-manage-10000s-of-kubernetes-clusters/, Dec. 2019.

[79] TSEITLIN, A. The Antifragile Organization. *Communications of the ACM 56*, 8 (Aug. 2013), 40–44.

[80] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)* (Apr. 2015).

[81] VMWARE. Introducing vSphere Lifecycle Management (vLCM). https://core.vmware.com/resource/introducing-vsphere-lifecycle-management-vlcm#section1.

[82] VMWARE. What is intent-based networking (IBN)? https://www.vmware.com/topics/glossary/content/intent-based-networking.html.

[83] XU, L., HUANG, J., HONG, S., ZHANG, J., AND GU, G. Attacking the Brain: Races in the SDN Control Plane. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)* (Aug. 2017).

[84] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[85] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)* (Apr. 2009).

[86] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)* (Apr. 2009).

[87] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[88] YUAN, X., AND YANG, J. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the 25th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'20)* (Mar. 2020).

[89] ZHANG, Y., YANG, J., JIN, Z., SETHI, U., RODRIGUES, K., LU, S., AND YUAN, D. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)* (Oct. 2021).

[90] ZOOKEEPER-OPERATOR-314. [BUG] Reading stale ZooKeeper cluster status can lead to undesired pod and PVC deletion. https://github.com/pravega/zookeeper-operator/issues/314, 2021.

ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory

Wonbae Kim[†] Chanyeol Park[§,¶] Dongui Kim[§,£] Hyeongjun Park[§]
Young-ri Choi[†] Alan Sussman[‡] Beomseok Nam[§]

UNIST[†] Naver[¶] Line[£] University of Maryland, College Park[‡] Sungkyunkwan University[§]

Abstract

Due to the latency difference between DRAM and non-volatile main memory (NVMM) and the limited capacity of DRAM, incoming writes are often stalled in LSM tree-based key-value stores. This paper presents *ListDB*, a write-optimized key-value store for NVMM to overcome the gap between DRAM and NVMM write latencies and thereby, resolve the write stall problem. The contribution of ListDB consists of three novel techniques: (i) byte-addressable *Index-Unified Logging*, which incrementally converts write-ahead logs into SkipLists, (ii) *Braided SkipList*, a simple NUMA-aware SkipList that effectively reduces the NUMA effects of NVMM, and (iii) *Zipper Compaction*, which moves down the LSM-tree levels without copying key-value objects, but by merging SkipLists in place without blocking concurrent reads. Using the three techniques, ListDB makes background compaction fast enough to resolve the infamous write stall problem and shows 1.6x and 25x higher write throughputs than PACTree and Intel Pmem-RocksDB, respectively.

1 Introduction

Non-Volatile Main Memory (NVMM) is a new tier in the memory/storage hierarchy. NVMM has latency comparable to DRAM, but ensures non-volatility of data, similarly to secondary storage. Because NVMM is installed in the memory slot, it is byte-addressable and operates at memory bus speeds.

In order to locate and retrieve data from large datasets in NVMM, an efficient persistent index that takes into account the characteristics of NVMM is required. In the past few years, various NVMM-only persistent indexing structures [11,13,24,35,45,49,56,63] and hybrid DRAM+NVMM persistent indexing structures [38,40,49,63] have been proposed. In addition, several key-value stores that manage large datasets using such persistent indexes and background worker threads have been developed [12, 29, 30, 57, 60]. While NVMM-only indexing structures such as Fast and Fair B+tree [24], CCEH [45], and PACTree [32] provide orders of magnitude higher performance than their disk-based counterparts, their performance is still lower than a DRAM index because commercial NVMM products, e.g., Intel's Optane DC Persistent Memory Module, a.k.a., DCPMM [26], fall short of the performance of DRAM. Specifically, DCPMM has (i) latency higher than DRAM, (ii) bandwidth lower than DRAM, (iii) high sensitivity to NUMA effects, and (iv) a larger media access granularity (i.e., 256-byte *XPLine*) [62], which transforms a small write into a larger read-modify-write operation.

To benefit from DRAM performance and avoid the shortcomings of NVMM, the hybrid DRAM+NVMM indexing structures and key-value stores proposed in previous studies [12,49,57,60] place the complexity of indexing in volatile DRAM. In this work, we question whether such a hybrid approach that ignores the byte-addressability, keeps the entire index in DRAM, and uses NVMM only as log space is desirable, because it has two major limitations. First, the capacity of DRAM is small. If a dataset's index does not fit in small DRAM, or if DRAM is shared with the working sets of other processes, the existing hybrid DRAM+NVMM approaches may suffer from memory swapping of large indexes. Second, a volatile DRAM index needs to be reconstructed from scratch when recovering from a system failure. If a large number of key-value objects are stored without a persistent index that can survive system crashes, the recovery time can be significant. To improve the recovery performance, a volatile index can be periodically checkpointed [12]. However, such a periodic synchronous checkpointing results in very high tail latency because it blocks concurrent writes.

We advocate *asynchronous incremental checkpointing*, merging small, high-performance DRAM indexes into a persistent index in the background for data recovery. *ListDB* is a write-optimized LSM (log-structured merge) tree-based key-value store for NVMM. ListDB achieves high performance comparable to DRAM indexes, and prevents a DRAM index from growing indefinitely by flushing to NVMM at high throughput exceeding that of a DRAM index. ListDB buffers bulk insertions in a small DRAM index, and runs background *compaction* threads to incrementally checkpoint the buffered writes to NVMM without data copy. Instead, ListDB restruc-

tures log entries as a SkipList rather than flushing the entire volatile index to NVMM. Simultaneously, such SkipLists are merged in place, reducing NUMA effects, without blocking concurrent read queries.

Specifically, ListDB proposes the following three novel techniques - *Index-Unified Logging*, *Zipper Compaction*, and *Braided SkipList*. Our contributions are as follows.

- **Fast Write Buffer Flush:** ListDB unifies the write-ahead log with SkipList. Using *Index-Unified Logging* (IUL), ListDB writes each key-value object to NVMM only once, as a log entry. Taking advantage of NVMM's byte addressability, IUL converts a log entry into a SkipList element in a lazy manner, which masks the logging and MemTable flush overhead. Therefore, it makes the MemTable flush throughput higher than the write throughput of the DRAM index, thus resolving the write stall problem.
- **Reducing NUMA Effects:** *Braided SkipList* effectively reduces the number of remote NUMA node accesses by making the upper layer pointers point only to the SkipList elements on the same NUMA node.
- **Fast Compaction with In-Place Merge-Sort:** *Zipper compaction* merge-sorts two SkipLists in-place without blocking read operations. By avoiding copy, Zipper compaction alleviates the *write amplification* [21, 41, 53] problem and reduces the number of SkipLists fast and efficiently to improve read and recovery performance.

Our performance study shows that the write performance of ListDB outperforms state-of-the-art NVMM-based key-value stores. For read performance, ListDB relies on classic caching techniques.

The rest of the paper is organized as follows. In Section 2, we present the background and motivation. In Section 3, we present the design of ListDB. In Section 4, we compare the performance of ListDB against state-of-the-art key-value stores. Finally, we conclude the paper in Section 5.

2 Background and Motivation

2.1 Hybrid DRAM+NVMM Key-Value Store

Intel's Optane DCPMM is much faster than block device storage. However, its performance is still worse than that of DRAM in terms of latency, bandwidth, NUMA sensitivity, and access granularity [62]. Furthermore, byte-addressable persistency complicates failure-atomicity (i.e., reusability after a system crash) because the CPU cache replacement mechanism may evict dirty cachelines that are not ready to be persisted. When a system recovers, such prematurely written cachelines may corrupt data structures. To guarantee failure-atomicity despite such unexpected cacheline flushes, NVMM-only data structures carefully order machine instructions using memory fence instructions and call expensive `clflush` instructions frequently to persist dirty cachelines, which incurs significant overhead in NVMM [24, 39, 56]. To avoid this, several hybrid DRAM+NVMM indexing structures and key-value stores have been proposed. For example, NV-tree [63] and FP-tree [49] are variants of B+tree that store internal tree nodes in DRAM and leaf nodes in NVMM. The internal nodes are lost upon a system crash but can be reconstructed from persistent leaf nodes. With this approach, writes to internal nodes do not need to be failure-atomic.

FlatStore [12] takes a rather radical approach, i.e., NVMM is used only as a log space where key-value objects are appended in insert order rather than key order, whereas the index resides in DRAM. Therefore, FlatStore has to reconstruct a volatile index from persistent log entries after a system crash. To mitigate the expensive recovery overhead, FlatStore proposes to checkpoint the DRAM index onto NVMM periodically. However, a naive synchronous checkpointing, as in FlatStore, takes a global snapshot while blocking incoming writes, leading to unacceptably high tail latency.

2.2 Log-Structured Merge Tree

2.2.1 Asynchronous Incremental Checkpointing

A better approach is *asynchronous incremental checkpointing* [28], which checkpoints only the difference between the current checkpoint and the last checkpoint state. Log-Structured Merge (LSM) tree [47] is a classic index that consolidates checkpoint data over time [10, 17, 20, 33, 36, 48, 54].

2.2.2 Write in LSM Tree

An LSM tree buffers multiple write operations in an in-memory buffer space called *MemTable*, which sorts key-value objects using an ordered index such as SkipList [10, 17, 20, 33, 36, 48, 54]. Since a MemTable is volatile, a key-value object is written to a write-ahead log (WAL) for crash consistency before it is inserted into the MemTable. If the MemTable size exceeds a certain threshold, it is marked as immutable and a new MemTable is created so that the new MemTable can serve incoming clients' requests while a background thread transforms the immutable MemTable into a sorted array called SSTable (Sorted String Table), flushes it to disk, and then deletes the corresponding log entries. This design leverages the high performance of DRAM for random writes and the high sequential write bandwidth of block devices.

The key range of a MemTable is not disjoint with those of SSTables on disk. If a large number of MemTables are converted into SSTables and the overlap between SSTables increases, background threads merge-sort them to incrementally construct fewer, eventually into one large sorted array for fast search. This process, called *compaction*, is the most significant performance bottleneck because the same key-value object is repeatedly written to new SSTables [2, 9, 21, 29, 37, 41–43, 46, 53, 55].

2.2.3 Search in LSM Tree

For a read query, an LSM tree looks up a mutable MemTable, immutable MemTables, and then SSTables from level 0 to

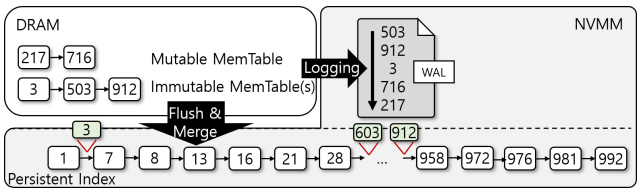


Figure 1: *Two-Level LSM Tree without Level 0 Buffer Indexes*

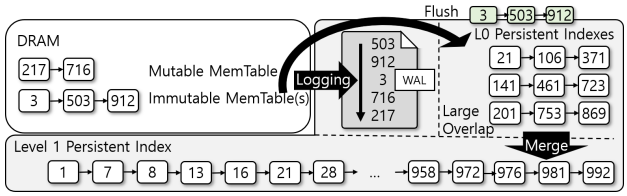


Figure 2: *Three-Level LSM Tree with Level 0 Buffer Indexes*

the upper levels, i.e., the recently stored objects are searched first. The search performance of LSM trees is affected by the degree of overlap between SSTables within and across levels because a read query searches all SSTables whose key range overlaps the search key until it finds a matched key. To reduce the overlap and improve the search performance, compaction threads merge-sort SSTables despite the high cost. Due to overlap and multiple levels, the read performance of LSM trees is worse than B+trees [27]. Nevertheless, LSM trees are more popular than B+trees in NoSQL systems because simple caching techniques can improve read performance. However, improving write performance is not easy.

2.2.4 Side Effect of Write Buffer: Write Stall

The in-memory MemTable is effective in buffering writes. However, despite buffering write bursts in the MemTable, tail latency can be very high if the workload is write-intensive because incoming writes can be blocked by *artificial governors* [31]. For instance, if compaction is slow, immutable MemTables will not be flushed to storage fast enough and the number of immutable MemTables will increase. Similarly, if SSTables are not merge-sorted quickly, the number of overlapping SSTables will increase, and search performance will degrade. Most LSM tree-based key-value stores [10, 17, 20, 33, 36, 48, 54] block clients from inserting new objects into the MemTable until compaction finishes and makes space for a new MemTable. This *write stall* problem occurs frequently in disk-based LSM tree-based key-value stores because of the high latency of the disk. If the write stall problem occurs, the insertion throughput is bounded by persistent storage performance, failing to benefit from the fast write buffer (DRAM) performance.

2.2.5 Write Amplification in LSM Trees

2.2.5.1 Multi-Level vs. Two-Level Compaction

As SSTables accumulate in storage, LSM trees perform compaction to merge-sort SSTables and reduce the overlap. Compaction is particularly expensive in disk-based key-value stores because they copy key-value objects between SSTable files. That is, compaction threads select a set of overlapping SSTables at level k and another set of SSTables that overlap at the next level $k+1$, and merge-sort them to create a new set of SSTables at level $k+1$. Such copy-based compaction allows concurrent read queries to access old SSTables while new SSTables are being created. However, copy-based compaction requires the same objects to be repeatedly copied to new SSTables. The number of times a key-value object is copied to a new file, called *write amplification* factor, has been reported to be as high as 40 [41, 53, 55]. The write amplification is particularly serious if key-value stores use *leveled compaction* and a large number of levels [53, 55]. The leveled compaction limits the number of SSTables per level and prevents any overlap between the SSTables at the same level.

NVMM allows byte-addressable updates. Therefore, there is an opportunity to avoid write amplification and improve compaction performance by replacing multiple levels of SSTables with a high-performance single-level persistent index. In particular, SLM-DB [29] uses two levels, i.e., MemTables and a single persistent B+tree in NVMM. Using the two-level design (shown in Figure 1), MemTables buffer multiple key-value objects and later insert them into a large persistent index in ascending order of keys, such that the large persistent index is traversed only once for multiple writes and it yields a higher write throughput than a single persistent index.

2.2.5.2 Decoupling Merge-Sort from Flush

The main problem with the two-level design is that the size of the persistent index affects the performance of merging volatile indexes into a persistent index, i.e., it fails to make write performance independent of NVMM performance. This is because MemTables are not flushed[1] *as-is*, but merge-sorted into the large, slow persistent index. Because NVMM has higher latency than DRAM, merge-sort throughput is much lower than insert throughput of volatile indexes, especially when the persistent index is large.

To alleviate this problem, most key-value stores including LevelDB [36] and RocksDB [54] employ an intermediate persistent buffer level (level 0, *L0*) in storage. That is, they flush MemTables to the intermediate buffer level without doing merge-sort. Figure 2 shows such a three-level design. By separating merge-sort from flush, MemTables can be flushed to NVMM faster; the flush throughput becomes independent of the database size.

A drawback of this design is that it results in a large number of overlapping SSTables, which hurts search performance. Given its poor indexing performance, the intermediate persistent buffer level does not appear to be very different from write-ahead log. Furthermore, key-value objects are written

[1] To avoid confusion with the cacheline flush instruction (e.g., `clflush`), writing a MemTable to NVMM is henceforth referred to as *flush*, and the cacheline flush is referred to as *persist*.

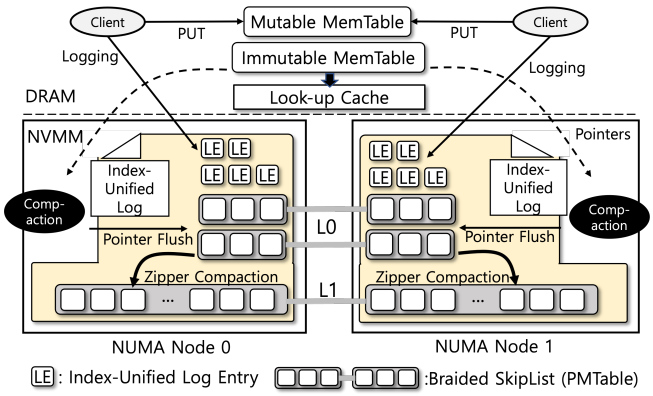


Figure 3: ListDB Architecture

to storage at least twice, i.e., once for WAL and once again for MemTable flush.

TRIAD [3], WiscKey [41], and FlatStore [12] prevent the same key-values (or just values) from being repeatedly written. TRIAD is particularly inspiring because it considers the commit log as an unsorted *L*0 SSTable. To enable efficient search in the unsorted *L*0 SSTables (the commit log), TRIAD creates a small index file for each *L*0 SSTable. The index file does not store keys and values, only the offsets for each object in sorted order of the keys. Although TRIAD reduces the I/O traffic, each MemTable flush creates an index file and calls the expensive `fsync()` to make it durable. However, given the high overlap between *L*0 SSTables and also the fact that *L*0 SSTables will be quickly merged into *L*1 SSTables, it is questionable whether a separate index file for each *L*0 SSTable should be created and persisted at a very high cost.

2.3 NUMA Effects

NVMM is more sensitive to NUMA effects than DRAM because of its lower bandwidth (1/6 for writes and 1/3 for reads) [16, 57, 62]. As such, state-of-the-art persistent indexes, such as FAST and FAIR B+tree [24] and CCEH [45] do not scale with the number of threads due to irregular cacheline accesses and NUMA effects [32, 57].

To mitigate the NUMA effects, Daase et al. [16] suggest limiting the number of write threads to 4-6 per socket. Nap [57] hides NUMA effects by overlaying a DRAM index on top of NVMM-resident indexes such that the DRAM index can absorb remote NUMA node accesses. However, data stored in NVMM is already in the memory address space, and NVMM has latency comparable to DRAM. Therefore, using DRAM as a fast cache layer over NVMM and copying data between DRAM and NVMM back and forth can be wasteful. For example, NVMM file systems such as EXT4-DAX and NOVA [61] do not use the page cache but directly access NVMM.

To mitigate NUMA effects in DRAM, various approaches, including *Delegation* with hash-based sharding [4, 6, 7, 44] and *Node Replication* (NR) [7] methods, have been investigated. In Delegation methods, a designated worker thread is assigned for all operations on a specific range of keys. Therefore, client threads have to communicate with worker threads and delegate operations using message passing. Due to the significant message passing overhead, Delegation performs sub-optimal, especially for lightweight tasks such as indexing operations [7]. Node Replication (NR) [7] implements a NUMA-aware shared log, which is used to replay the same operations for the data structures replicated across NUMA nodes. However, this consumes memory for replicating the same data structure across multiple NUMA nodes. Besides, the performance falters due to cross-node communication, as the number of NUMA nodes increases [7]. Considering that the bandwidth of Optane DCPMM is much lower than that of DRAM [62], replication can aggravate the low bandwidth problem.

3 Design of ListDB

ListDB is a write-optimized key-value store with an LSM tree structure that resolves the write stall problem. In this section, detailed descriptions of ListDB's key designs are provided. First, the overall architecture of ListDB is presented (§3.1). Then, its key designs, i.e., *Index-Unified Logging* (§3.2), NUMA-aware *Braided SkipList* (§3.3), in-place *Zipper Compaction* (§3.4), *lookup cache* (§3.5), and recovery algorithm (§3.6) are presented.

3.1 Three-Level Architecture

Figure 3 shows the three-level architecture of ListDB-volatile MemTables, and *L*0 and *L*1 Persistent MemTables (*PMTables*). MemTables and PMTables are essentially the same SkipLists, but the node structure of PMTable has additional metadata that MemTable does not need because PMTable is a data structure transformed from the write-ahead log. ListDB uses SkipList as the core data structure for all levels because it enables byte-addressable in-place merge-sort and avoids the *write amplification* problem [21, 41, 53], as will be presented throughout the paper.

ListDB employs an intermediate persistent buffer level - *L*0 (level 0) in NVMM. With level 0, a MemTable is flushed to NVMM without being merge-sorted, making the flush throughput independent of the next level persistent index size. MemTables accumulated at *L*0 (*L*0 PMTables) are gradually merged into the large *L*1 PMTable by compaction. To manage multiple PMTables, ListDB uses a metadata object called MANIFEST to point to the beginning of each SkipList.

3.2 Index-Unified Logging

ListDB aims to flush MemTables to NVMM without copying key-value objects. As discussed in Section 2.2.5.2, all key-value objects in MemTables are already persisted in the commit log in NVMM [3]. Besides, *L*0 indexes are known to have very poor indexing performance due to large overlap.

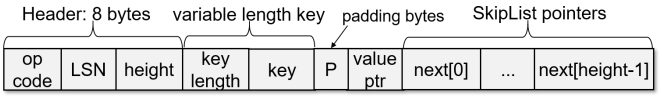


Figure 4: Index-Unified Log Entry Layout

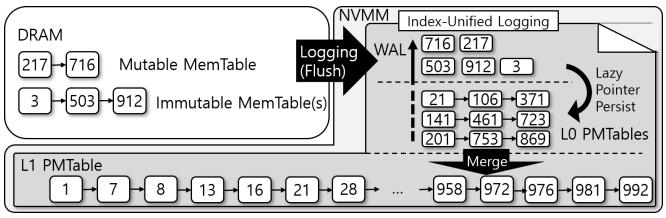


Figure 5: Index-Unified Logging

3.2.1 Conversion of IUL into SkipList

Index-Unified Logging (IUL) unifies write-ahead log entries and SkipList elements by allocating and writing log entries in the form of SkipList elements. Figure 4 shows the structure of an IUL entry, which serves both as a log entry and as a SkipList element. When a key-value object is inserted into a MemTable, the object and its metadata (i.e., operation code `op_code` and log sequence number `LSN`) are written and persisted as a log entry in NVMM with SkipList pointers initialized to NULL (Algorithm 1). Later, when a compaction thread flushes its corresponding MemTable from DRAM, the log entry is converted into a SkipList (*L*0 PMTable) element, reusing the key and value stored in the log entry.

The information that *L*0 PMTable needs, but the log does not have, is the sorted order of keys, which is managed as SkipList pointers in MemTables. When converting the log into an *L*0 PMTable, the addresses of the corresponding MemTable elements are simply translated into NVMM addresses, i.e., the log entry offsets, as shown in Algorithm 2. When the SkipList pointers in IUL entries are set to NVMM addresses, the IUL entries become SkipList elements.

Finally, the MANIFEST is updated to validate the new *L*0 PMTable and invalidate the immutable MemTable in a failure-atomic transaction.

3.2.2 MemTable Flush without `clflush`

When writing SkipList pointers to log entries, there is no need to call persist instructions (e.g.,`clflush`) because the key-value objects are already persistent in the log, and because the order of keys can be recovered without difficulty in case of a crash. Instead of explicitly persisting cachelines for updated pointers, Index-Unified Logging leaves that to the CPU cache replacement mechanism, i.e., it waits until the CPU evicts updated pointers from its cache. Through the CPU cache replacement mechanism, multiple pointer updates to the same 256-byte XPLine can be buffered and batched. That is, each 8-byte small write is not eagerly transformed into a 256-byte read-modify-write operation. Not only does it defer the read-modify-write problem, but also prevents background compaction threads from being affected by the read-modify-write problem and high NVMM write latency.

Algorithm 1 *Put(kvObject)*

```
1:  mutex.lock();
2:  iul_entry ← iul_tail;
3:  iul_entry.LSN ← GetNextLSN(); /* log sequence number */
4:  iul_entry.height ← RandomHeight(); /* SkipList element height */
5:  iul_tail ← iul_tail + sizeof(kvObject) + height*8 + 8;
6:  mutex.unlock();
7:  iul_entry.op_code ← OP_INSERT; /* operation type (insert, delete) */
8:  iul_entry.kvObject ← kvObject;
9:  iul_entry.next[0..height] ← NULL; /* initialize pointers */
10: pmem_persist(iul_entry, sizeof(iul_entry)); /* calls clwb */
11: memTable.Insert((SkipListElement)iul_entry); // classic SkipList insert
```

Algorithm 2 *FlushImmutableMemTable(memTable)*

```
1:  element ← memTable.head[0].next[0]; // smallest MemTable element
2:  while element≠NULL do
3:      L0_element ← element.iul_address;
4:      lookup_cache.Insert(L0_element);
5:      for layer ← 0; layer < element.height; layer++ do
6:          L0_element.next[layer] ← element.next[layer].iul_address;
7:          /* no need to persist */
8:      end for
9:  end while
10: new_L0.iul_address ← memTable.head[0].next[0].iul_address;
11: new_L0.next ← MANIFEST.L0List().GetFront();
12: MANIFEST.L0List().PushFront(new_L0); /* CAS */
13: freeMemTable(memTable);
```

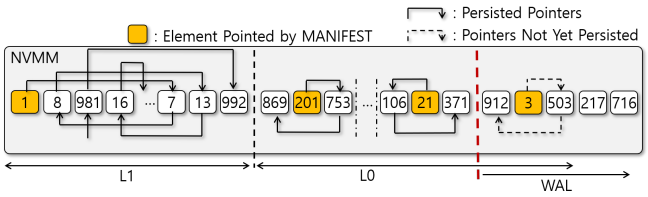
3.2.3 Walk-Through Example

Let us walk through MemTable flush illustrated in Figure 5. Suppose foreground client threads insert keys into the currently mutable MemTable in the order of 503, 912, and 3. Each client thread persists the object, its metadata, and NULL pointers in the log before it commits. Then, a background thread marks the MemTable as immutable and creates a new MemTable. Client threads insert two more keys, 716 and 217, into the new mutable MemTable.

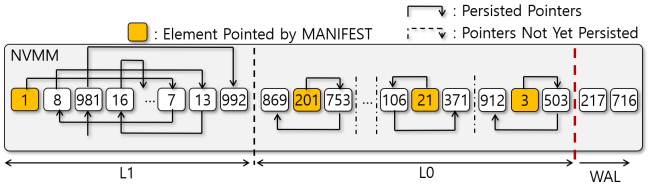
When a background compaction thread flushes the immutable MemTable, i.e., (3, 503, 912), the pointers of each MemTable element are simply translated into the IUL offsets of the corresponding log entries and the NULL pointers are replaced with the IUL offsets so that the log entries become a SkipList, as shown in Figure 6(a). As described in Section 3.2.2, the updated pointers in the new *L*0 PMTable may remain in the CPU cache and may be lost upon a system crash, but the pointers are not required for crash consistency.

3.2.4 Checkpointing *L*0 PMTable

Although the log entries are now converted to *L*0 PMTable elements, the boundary between logging space and *L*0 PMTable space (denoted as a thick dotted line in Figure 6(a)) has not moved, because it is not guaranteed that the pointers of the new *L*0 PMTable are persistent. The boundary can only move if `clflush` instructions are explicitly called for the updated pointers. In our implementation, a background thread persists dirty cachelines for *L*0 PMTables in batches. This op-



(a) NVMM Layout Before Checkpointing



(b) NVMM Layout After Checkpointing

Figure 6: NVMM Layout of Index-Unified Logging

eration is referred to as *checkpointing*. Figure 6(b) shows the NVMM layout after the pointers are explicitly persisted. Once a PMTable is checkpointed, it is possible to move the boundary of the logging space to reduce the number of log entries to recover, as shown in Figure 6(b).

3.2.4.1 Lazy Group Checkpointing

Checkpointing reduces recovery time. However, ListDB defers checkpointing as much as possible, because calling `clflush` instructions is very expensive. Even if *L*0 PMTables are not persisted at all, it does not affect crash consistency because all the elements in all *L*0 PMTables will be treated as log entries if the system crashes, and the key order of *L*0 PMTable elements can be reconstructed from the log.

In our implementation, multiple *L*0 PMTables are grouped and dirty cachelines for them are persisted in batches. We call this *lazy group checkpointing*. Note that there is a trade-off between lazy group checkpointing and recovery time. Infrequent checkpointing increases the log size and it takes longer to recover. In contrast, if checkpointing frequency is high, recovery will be fast, but flush throughput degrades.

Zipper compaction, which will be described in Section 3.4, persists pointers fast enough to prevent the number of *L*0 PMTables from increasing. That is, even if IUL does not persist any *L*0 PMTable, Zipper compaction persists pointers fast when merging an *L*0 PMTable into the *L*1 PMTable, and the recovery time of IUL is much shorter than synchronous checkpointing, as will be shown in Section 4.

3.3 NUMA Effects for SkipList

ListDB employs a NUMA-aware data structure, which is more scalable and effective in minimizing NUMA interconnect contention than Delegation and Node Replication [7].

3.3.1 NUMA-aware Braided SkipList

A SkipList has the invariant that the list at each layer[2] is a sorted sub-list of the bottom layer [52]. Unless this invariant

[2]To avoid confusion with the *level* of LSM trees, the level of SkipList will be referred to as *layer*.

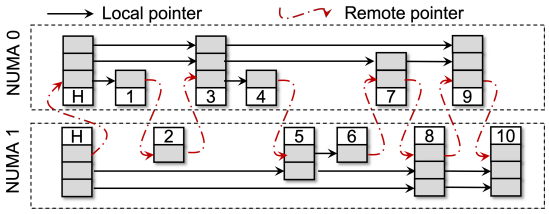


Figure 7: NUMA-aware Braided SkipList

is violated, correct search results are guaranteed because the upper layer pointers are probabilistic shortcuts, which do not affect the correctness of search results. However, an upper layer does not need to be a sub-list of the next layer, as long as it is a sub-list of the bottom layer. Even if a search does not find a key closer to the search key in an upper layer, the search falls back to a lower layer and eventually to the bottom layer which contains all sorted keys.

The Braided SkipList of ListDB leverages this property to mitigate NUMA effects in a simple and effective way. Upper layer pointers ignore SkipList elements in remote NUMA nodes; i.e., upper layer pointers of each element point to an element with a larger key in the same NUMA node. Compared to NUMA-oblivious conventional SkipLists, Braided SkipList reduces the number of remote memory accesses to $1/N$, where N is the number of NUMA nodes, as will be shown in Section 4.

Figure 7 illustrates an example (The upper layers in NUMA node 1 are illustrated upside down for ease of presentation). Observe that the second layer pointer of element `3` on NUMA node 0 points to element `7` on the same NUMA node, instead of element `5` on NUMA node 1. Nonetheless, a correct search is guaranteed. For example, suppose a client thread running on NUMA node 0 searches for element `5`. It will follow the top layer to element `3`, then `9`. Since `9` is greater, the thread moves down one layer in element `3`, and then the search visits element `7`. Since `7` is greater than `5`, the thread moves down again and follows the bottom layer pointer to element `4`. Since the search key is greater than `4`, it follows the bottom layer to a remote SkipList element `5`. The search then completes.

In our implementation of Braided SkipList, a NUMA ID is embedded in the extra 16 bits of the 64-bit virtual address, as in *pointer swizzling* [59], such that it can use 8-byte atomic instructions instead of expensive PMDK transactions [50]. For direct reference, Braided SkipList restores the virtual memory address of a SkipList element by masking the extra 16 bits.

3.4 Zipper Compaction

With byte-addressable NVMM, Zipper compaction merge-sorts *L*0 and *L*1 PMTables in-place by only updating pointers, but without blocking concurrent read queries. The in-place merge-sort avoids write amplification, thus it improves the compaction throughput.

Leveraging the SkipList invariant (§3.3.1), various lock-free SkipLists have been studied in the literature [22, 23], and the Java™ SE `ConcurrentSkipListMap` class has been

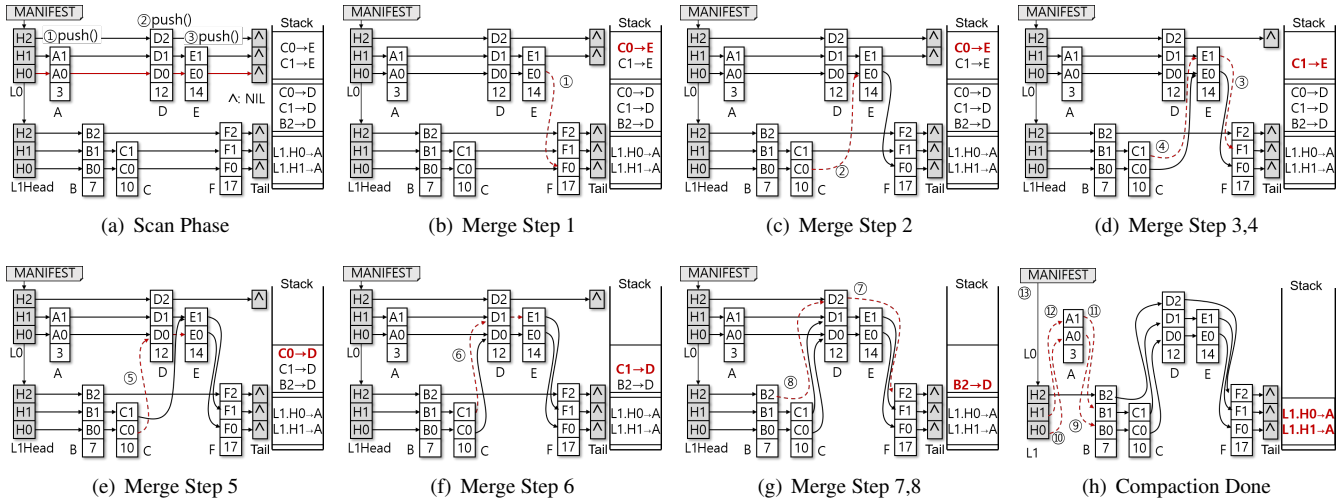


Figure 8: *Zipper Compaction: Merging SkipLists from Tail to Head*

shown to perform well in practice [34]. Zipper compaction algorithm allows concurrent read operations to access $L0$ and $L1$ PMTables while merging them without violating the invariant of SkipList.

A classic lock-free SkipList avoids locks for multiple writers. In contrast, ListDB does not perform concurrent writes; the only writers are the compaction threads, and ListDB coordinates them to avoid write-write conflicts. For parallelism, multiple compaction threads write to disjoint *shards*. A shard is a disjoint key range from an element with the maximum height to the next element with the maximum height in $L1$ PMTable. To merge $L0$ elements into $L1$, a compaction thread must acquire a lock on the corresponding shard.

Zipper compaction proceeds in two phases; (i) a forward *scan* from head to tail and (ii) a backward *merge* from tail to head, hence the name. To guarantee correct search results without blocking concurrent readers, $L0$ PMTable elements are merged into $L1$ PMTable from tail to head while concurrent read operations are traversing them from head to tail.

3.4.1 Scan Phase

In the forward scan phase, a compaction thread traverses $L0$ and $L1$ PMTables from head to tail and determines where each $L0$ PMTable element should be inserted in the $L1$ PMTable. However, in this phase, it does not make any change to the PMTables but pushes necessary pointer updates on a stack. The backward merge phase pops the stack to apply and persist the updates to the $L1$ PMTable.

The scan phase follows the bottom layer of $L0$ PMTable. For each $L0$ element, it searches the $L1$ PMTable to find where to insert the $L0$ element. For this, it keeps track of the rightmost element smaller than the current search key ($L0$ element) in each layer to avoid repeatedly traversing $L1$ PMTable. Since keys are sorted in both PMTables, the next larger key in $L0$ PMTable can reuse the previous rightmost elements, and backtrack to the top-layer rightmost element for the next search. Therefore, the complexity of the scan phase is $O(n_0 + \log n_1)$ where n_0 and n_1 are the sizes of $L0$ and $L1$ PMTables, respectively.

Algorithm 3 shows the pseudo-code of Zipper compaction. For NUMA-aware Braided SkipLists, Zipper compaction requires a two-dimensional array - `rightmost[numa_id][layer]` to keep as many rightmost elements in each layer as the number of NUMA nodes for Braided SkipList. But, note that a Braided SkipList element does not need more pointers than a conventional SkipList element as it embeds NUMA node ID in the 8-byte address.

Figure 8(a) shows an example of Zipper scan. For ease of presentation, all SkipList elements are assumed to be on the same NUMA node. The first element `A` in $L0$ will be placed in the first position in $L1$. Hence, `H0` and `H1` of the head element in $L1$ are the current rightmost pointers that need to be updated for `A`. This information is stored on the stack. Note that `A0` and `A1` need to point to `B`, but they are not pushed onto the stack because `B` is pointed by the current rightmost elements that are already pushed on the stack. Each $L0$ element is inserted between two $L1$ elements and only the previous (i.e., rightmost) element in each layer needs to be pushed on the stack because the next element can be found from the previous elements. Next, the scan phase visits the second element `D` in $L0$ and searches $L1$. Inserting `D` requires updating `B2`, `C1`, and `C0`. Again, they are pushed onto the stack. Finally, it visits the last element `E` in $L0$ and searches $L1$. Note that $L1$ PMTable has not changed and the current rightmost pointers are still `B2`, `C1`, and `C0`. Thus, the scan phase pushes `C1` and `C0` on the stack to make them point to `E`.

3.4.2 Merge Phase

The merge phase applies pointer updates from tail to head. When a compaction thread pops a pointer update $X_N \rightarrow Y$ from the stack, the Nth layer pointer in element Y is updated to the current value of X_N. Then, X_N is set to the address of Y.

Algorithm 3 *BraidedZipperCompaction(L0SkipList, L1SkipList)*

```
1: LogZipperCompactionBegin(L0SkipList); // micro-logging
2: L0_element ← L0SkipList.head[0].next[0]; // smallest L0 element
3: local_numa_id ← DecodeNumaId(L0_element); // not always 0
4: for i ← 0; i < NumNUMA; i++ do
5:     rightmost[i][] ← L1SkipList.head[i].next[]; // array copy
6: end for
7: bottom_L1_element ← L1SkipList.head[0].next[0];
8: L1_element ← L1SkipList.head[local_numa_id]; // local head
9: // I. scan phase: from head to tail
10: while L0_element≠NULL do
11:     // NUMA-aware local search for upper layer pointers
12:     for layer ← L0_element.height−1; layer > 0; layer−− do
13:         while L1_element.next[layer] ≠ NULL &&
                  L1_element.next[layer].key < L0_element.key do
14:             L1_element ← L1_element.next[layer];
15:             // update the rightmost for the current layer
16:             rightmost[local_numa_id][layer] ← L1_element;
17:         end while
18:     end for
19:     // NUMA-oblivious search for bottom-layer pointer, i.e., layer = 0
20:     L1_element ← bottom_L1_element;
21:     <<same while loop with line 13–17 >>
22:     // push an array of NUMA local upper-layer pointers and a NUMA-
        oblivious bottom layer pointer
23:     stack.push(L0_element, rightmost[local_numa_id][]);
24:     // fetch the next L0_element and update local NUMA ID
25:     L0_element ← L0_element.next[0];
26:     local_numa_id ← DecodeNumaId(L0_element);
27:     bottom_L1_element ← L1_element;
28:     L1_element ← rightmost[local_numa_id][L0_element.height−1];
29: end while
30: // II. merge phase: from tail to head
31: while stack is not empty do
32:     (L0_element, rightmost2update[]) ← stack.pop();
33:     for layer ← 0; layer < L0_element.height; layer++ do
34:         // Pop and apply the updates without worries about NUMA IDs
35:         L0_element.next[layer] ← rightmost2update[layer].next[layer];
36:         if layer = 0 then
37:             persist(L0_element.next[layer]);
38:         end if
39:         rightmost2update[layer].next[layer] ← L0_element;
40:         if layer = 0 then
41:             persist(rightmost2update[layer].next[layer]);
42:         end if
43:     end for
44:     second_chance_cache.Insert(L0_element);
45: end while
46: MANIFEST.L0List().PopBack(); /* CAS */
47: LogZipperCompactionDone(L0SkipList); // micro-logging
```

In the example, shown in Figure 8(b), the compaction thread pops `C0`→`E` and sets `E0` to `F`, which is the current value of `C0`. At this point, the upper layer pointer of element `E` (`E1`) is not pointing to element `F`. However, as described earlier, upper layer pointers are probabilistic shortcuts, which do not affect the correctness of search. Therefore, there is no need to update `E0` and `E1` atomically. In the next step, shown in Figure 8(c), the compaction thread sets `C0` to the address of `E`. In the next step, shown in Figure 8(d), the compaction thread pops `C1`→`E`, sets `E1` to `F`, and makes `C1` point to `E`. Each pointer update is removed from the stack one by one, and is applied in order, as shown in Figures 8(e), 8(f), 8(g), and 8(h). Zipper compaction assumes 8-byte pointer updates are atomic. To make the updates failure-atomic, it persists each bottom layer update immediately using memory fence and cacheline flush instructions. In the final step, the compaction thread deletes the head element of *L*0 PMTable from the MANIFEST object, thus completing compaction.

3.4.3 Lock-Free Search

Zipper compaction does not violate the correctness of concurrent search, i.e., a read thread will not miss its target SkipList element without acquiring a lock. This is because a read thread accesses PMTables from head to tail and from *L*0 to *L*1, whereas a compaction thread merges them from tail to head. During Zipper compaction, every element is guaranteed to be pointed by at least one head. Consider the example shown in Figure 8, which shows how a sequence of atomic store instructions merges the two example SkipLists. Even if a concurrent read thread accesses the PMTables in any state shown in Figure 8, it returns a correct result.

The algorithm remains correct even if a read thread is suspended during compaction thread is making changes to SkipLists. For example, suppose a read is suspended while accessing an *L*0 element. When it resumes, the element might have been merged into *L*1. When the read thread wakes up, it will continue traversing to the tail if it does not find the search key. Once it reaches the tail, it is done with *L*0 and will start searching *L*1, into which *L*0 elements have been merged. Consequently, the read thread might visit the same elements multiple times, but it will never miss the element it is searching. Multiple visits might hurt search performance. To avoid this, a read stops searching the *L*0 if it detects the level of the current element is *L*1.

3.4.4 Updates and Deletes

An update in LSM trees duplicates the same key because writes are buffered in MemTables and gradually flushed to the last level. ListDB does not eagerly delete the older version in *L*1. Instead, when a compaction thread scans *L*0 and *L*1 levels for Zipper compaction, it marks the older version in *L*1 obsolete. Similarly, a delete in ListDB does not physically delete an object but inserts a *key-delete* object into the MemTable. If an LSM tree physically deletes the most recent version of a key from MemTables or *L*0 PMTables, older versions of the key will come back to life. Zipper compaction places a more recent key-value or key-delete object before its corresponding old objects. Therefore, a read query always accesses the more recent object before older ones, and thus returns a correct search result.

3.4.5 Fragmentation and Garbage Collection

Using `libpmemobj` library [50], ListDB allocates and deallocates a *memory chunk* (e.g., 8 MB) for multiple IUL entries in PMDK's failure-atomic transaction so that the number of calls to expensive PMDK transactions can be reduced. ListDB deallocates a memory chunk if all elements in the chunk are marked obsolete or deleted. Note that ListDB does not relocate SkipList elements for garbage collection. To address

the lasting fragmentation, a compaction thread may perform CoW-based garbage collection. We leave this optimization for our future work.

Memory management for lock-free data structures is a hard problem because there is no easy way to detect whether deallocated memory space is still being accessed by concurrent reads [5, 14, 19]. ListDB employs a simple epoch-based reclamation [14]; ListDB does not deallocate memory chunk immediately but waits long enough for short-lived read queries to finish accessing the deallocated memory chunk. A background garbage collection thread periodically checks and reclaims a memory chunk if all objects in the memory chunk are obsolete or deleted. For obsolete objects, the garbage collection thread checks its newer version's LSN. If it is also old enough, it considers the obsolete objects are not accessed by any reads, removes them from $L1$ PMTable, and physically deallocates the memory chunk.

3.4.6 Linearizability

Theorem 1. *Zipper compaction is linearizable with a single writer and multiple readers.*

Proof. For some element e, there is a single *linearization point* [23] for a writer when its level changes from $L0$ to $L1$, by atomic update of the bottom-layer next pointer. We denote this linearization point as $e.merge\_to\_L1$.

There are two linearization points $e.search\_L0$ and $e.search\_L1$ for a reader, as it searches $L0$ and then $L1$ PMTable in order. Let $a \rightarrow b$ if an event a happens before another event b. There are three cases to consider.

1. $e.merge\_to\_L1 \rightarrow e.search\_L0 \rightarrow e.search\_L1$
2. $e.search\_L0 \rightarrow e.merge\_to\_L1 \rightarrow e.search\_L1$
3. $e.search\_L0 \rightarrow e.search\_L1 \rightarrow e.merge\_to\_L1$

In case 1, $e.search\_L1$ will find e in $L1$. In case 2, $e.search\_L0$ will find e in $L0$. If the search does not stop after finding e in $L0$, $e.search\_L1$ will also find e in $L1$. In case 3, similarly, $e.search\_L0$ will find e. Since all three cases succeed in finding e, Zipper compaction is linearizable, meaning a read always succeeds in finding an element if the element was inserted by a committed write transaction, regardless of whether the element is in $L0$ or $L1$ PMTable. □

3.5 Look-up Cache

ListDB requires that a read query accesses at least two indexes, i.e., a mutable MemTable and $L1$ PMTable. Therefore, the read throughput of ListDB is significantly lower than a highly-optimized persistent B+tree, as we show in Section 4.

To mitigate this problem, ListDB uses a *lookup cache* in DRAM. Flushing a MemTable hashes each element into a fixed-sized static hash table. Unlike disk-based designs, the lookup cache does not duplicate the element in it, but only stores its NVMM address because the element in NVMM is already in the memory address space and its address never changes. Hence, regardless of the level at which the PMTable element is present, the lookup cache can locate the PMTable element. SkipList pointers are frequently updated by compaction threads in ListDB. By caching immutable addresses, not mutable content, the lookup cache can avoid frequent cache invalidation. If a hash collision occurs on a bucket, the old address is overwritten (i.e., FIFO replacement policy).

Algorithm 4 *Get(key)*

```
1: iter ← MANIFEST.GetTableIterator();
2: table ← iter.GetTable(); // get mutable MemTable
3: while table ≠ NULL && table.IsPMTable() = false do
4:     value ← table.Search(key); // SkipList lookup
5:     if value ≠ NULL then
6:         return value; // Found: return value
7:     end if
8:     table ← (++iter).GetTable(); // immutable MemTables
9: end while
10: /* L0 Cache Lookup */
11: cached ← lookup_cache.Lookup(key);
12: if cached ≠ NULL && cached.GetElement().key = key then
13:     return cached.GetElement().value;
14: end if
15: /* L0 Search */
16: while table ≠ NULL && table.Level() = 0 do
17:     value ← table.Search(key); // SkipList lookup
18:     if value ≠ NULL then
19:         return value; // Found: return value
20:     end if
21:     table ← (++iter).GetTable(); // L0 PMTables
22: end while
23: /* L1 Search */
24: rightmost ← second_chance_cache.Lookup(key);
25: value ← table.SearchFromElement(key, rightmost);
26: if value ≠ NULL then
27:     return value; // Found: return value
28: end if
29: return NOT_FOUND;
```

ListDB constructs a SkipList in DRAM as a second chance lookup cache for tall elements evicted from the hash table. The purpose of the second chance lookup cache is to accelerate PMTable search. Even if a key is not found in the second chance cache, a query can start the search from the closest PMTable element found in the cache. Algorithm 4 shows how a read query uses the lookup caches. Suppose a read searches for key 100 but finds element 85 is the closest smaller element in $L1$. Then, the search continues from element 85 in $L1$ PMTable instead of the beginning of $L1$. ListDB does not use the second chance lookup cache for $L0$ search because small $L0$ PMTables are merged into $L1$ fast, and $L0$ elements are mostly cached in the lookup hash table, The second chance lookup cache uses the SIZE replacement policy [58], i.e., it compares heights and evicts elements with shorter heights.

3.6 Recovery

A system may crash while $L0$ and $L1$ PMTables are being merged by Zipper compaction. To recover from such failures, a compaction thread performs micro-logging to keep track of which $L0$ PMTable is being merged into $L1$ PMTable. When a system restarts, ListDB checks the compaction log to redo unfinished compactions. For redo operations, Zipper compaction has to check duplicate entries since many entries in the tail of

Algorithm 5 *RecoverDB()*

```
1:  ScheduleUnfinishedZipperCompactionJob();
2:  curr_table ← NULL;
3:  while log_iter.Valid() do
4:      iul_entry ← log_iter.GetIULEntry();
5:      table_id ← GetTableIdByLSN(iul_entry.LSN);
6:      if curr_table = NULL || table_id ≠ curr_table.Id() then
7:          curr_table ← MANIFEST.GetTableById(table_id);
8:          curr_table.ResetSkipListHead();
9:      end if
10:     curr_table.InsertEntry(iul_entry); /* SkipList Insert */
11:     log_iter.Next(); /* from old to latest */
12: end while
```

*L*0 PMTable can be shared with *L*1 PMTable.

The recovery algorithm of ListDB, shown in Algorithm 5, is similar to that of conventional LSM trees. First, a recovery process locates the boundary of WAL, which is recorded by compaction threads in the compaction log. Then, it sorts log entries and restores *L*0 PMTables. At this point, the system returns to the normal execution mode and starts processing clients' queries. Compaction between *L*0 and *L*1 will be done in the background as normal.

As for the lookup cache, ListDB can process clients' queries without restoring the cache although the search performance will be poor until the cache is populated. By avoiding the reconstruction of DRAM cache and index, the recovery performance of ListDB is superior to synchronous checkpointing [12], as we show in Section 4.

4 Evaluation

4.1 Experimental Setup

Experiments are conducted on a four-socket NUMA server with Intel Xeon Gold 5215 CPU (2.50 GHz, 20 vCPUs) per socket, 256 GB of DDR4 DRAM (16x 16 GB), and 2 TB (16x 128 GB) Optane DCPMM (4 DCPMM's and 4 DRAM's per each socket) in app-direct mode. Our testbed server supports only the *directory coherence* protocol, but not *snoop* protocol, despite its known NUMA bandwidth meltdown issues [32].

All implementations are compiled using gcc 7.5.0 with `-O3` optimization. Using PMDK [50], ListDB creates an auto-growing directory-based persistent memory poolset (`pmempool`) on each NUMA node [50]. For NUMA-oblivious designs, we use the device mapper to create a single persistent memory poolset interleaved on four sockets.

We evaluate the performance of ListDB[3] using two individual sets of experiments. First, the performance effects of each part of the design of ListDB are quantified. Second, the performance of ListDB is compared against that of state-of-the-art persistent indexes, including FAST and FAIR B+tree [24] and PACTree [32], and LSM tree-based key-value stores for NVMM, i.e., NoveLSM [30], SLM-DB [29], and Intel's industry-optimized Pmem-RocksDB [51]. Pmem-RocksDB [51] is a variant of RocksDB for NVMM that Intel has optimized in two respects. First, Pmem-RocksDB separates keys and values to mitigate write amplification issues, as in WiscKey [41]. Second, Pmem-RocksDB mmaps SSTables and writes directly to NVMM by using non-temporal stores (i.e., `ntstore`) to bypass the cache hierarchy and eliminate context switching.

[3] Source code is available at http://github.com/DICL/listdb.

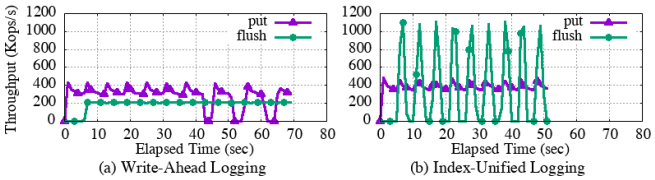


Figure 9: *Low Flush Throughput Results in Write Stalls*

Our experiments use YCSB [15] and the Facebook benchmark [8]. The Facebook benchmark generates more realistic workloads than YCSB as it emulates real-world RocksDB workloads in Facebook/Meta datacenters. Specifically, the Facebook benchmark adds mathematical models (e.g., sine distribution) to db_bench [18] such that it can vary key sizes, value sizes, and query arrival rates over time.

4.2 Evaluation of Index-Unified Logging

4.2.1 IUL vs. WAL: Flush Throughput

This section compares the performance effect of IUL to standard WAL with respect to write stalls. For the experiments shown in Figure 9, a single YCSB [15] client thread (Load A) and a single compaction thread are used to evaluate how fast a MemTable absorbs bursts of 20 million writes (8-byte key and 8-byte value objects), and how fast a single background compaction thread flushes MemTables to NVMM. To prevent memory usage from increasing indefinitely, the maximum number of immutable MemTables is set to 4. Zipper compaction threads are disabled to evaluate only the effect of IUL, i.e., *L*1 is not used. `put` denotes the client's query processing throughput over time (i.e., the number of records inserted into MemTables per second), and `flush` denotes how many records are flushed from MemTables to NVMM by the compaction thread.

Figure 9 (a) shows that with standard WAL, `put` throughput is higher than `flush` throughput because inserting key-value objects into a SkipList in DRAM is much faster than flushing (i.e., copying key-value objects from DRAM to NVMM) and persisting a SkipList in NVMM. Each spike in `put` throughput indicates that a new empty mutable MemTable was created; it takes about 5 seconds to fill a 64 MB MemTable. In 40 seconds, the number of MemTables exceeds the threshold, and subsequent writes are blocked. Even if the threshold is set to a higher value than four, it is only a matter of time before a write is stalled, because `flush` throughput is lower than `put` throughput.

In contrast, Figure 9 (b) shows that with IUL, `flush` throughput is much higher than `put` throughput. `flush` throughput of IUL fluctuates because each flush takes less time than filling a MemTable, i.e., the compaction thread be-

(a) Load A (b) Workload A (c) Workload B (d) Workload C (e) Workload D

Figure 10: Performance Effect of Index-Unified Logging

comes idle. This high `flush` throughput is because IUL does not copy key-value objects from DRAM to NVMM and does not call cacheline flush instructions. So, write stalls do not occur and the compaction thread often becomes idle, allowing the CPU to perform other work.

4.2.2 Evaluation of IUL using YCSB

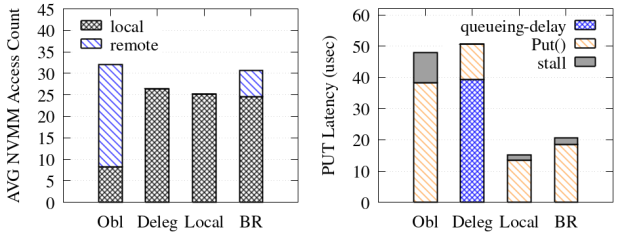
The experiments shown in Figure 10 compare the performance of IUL to standard WAL with varying the number of client threads for YCSB workloads. The number of background compaction threads is set to half the number of client threads. The MemTable size and the maximum memory usage for MemTables are set to 256 MB and 1 GB, respectively, i.e., a maximum of 4 MemTables are allowed. We set the lookup cache size to 1 GB (979 MB hash-based lookup cache and 45 MB for the second chance lookup cache). For the experiments, Braided SkipList and Zipper compaction are enabled so that $L0$ PMTables are merged into $L1$ PMTable and read queries can run faster. The `Load A` workload populates the database with 100 million records (8-byte keys and 8-byte values). All other workloads submit 100 million queries each.

Figure 10(a) shows that increasing the number of client threads increases the write throughput of both logging methods, up to 80 threads. With 80 client threads, the throughput of IUL (14.513 million ops/sec) is approximately 1.8x higher than that of WAL (8.101 million ops/sec). However, when the number of client threads exceeds the number of logical cores throughput degrades due to the high overcommit rate. That is, 100 client threads and 50 background compaction threads compete for 80 logical cores. Still, the throughput of IUL is 99% higher than WAL.

For Workload B (95% reads), Workload C (100% reads), and workload D (read latest), WAL has similar or slightly better performance than IUL because WAL does copy-on-writes to store records in ascending order of keys, and read operations benefit from higher memory access locality than IUL. Nevertheless, IUL outperforms WAL in Workload A (50:50 Read:Write) due to its better write performance.

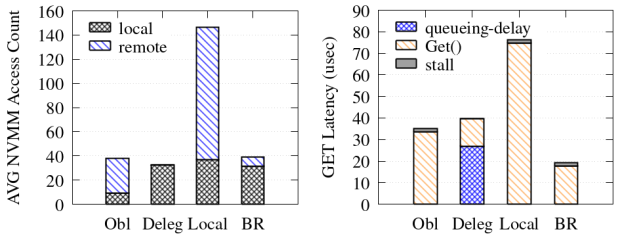
4.3 Evaluation of Braided SkipList

This section evaluates NUMA effects in NVMM using a single PMTable. The performance of the NUMA-aware Braided SkipList (denoted as `BR`) is compared with three other methods that were discussed in Section 2.3; i.e., (i) NUMA-oblivious SkipList (denoted as `Obl`), (ii) delegating client



(a) Memory Access Count (b) Response Time Breakdown

Figure 11: PUT Performance (80 Clients)



(a) Memory Access Count (b) Response Time Breakdown

Figure 12: GET Performance (80 Clients)

queries to a worker thread, using shared memory (denoted as `Deleg`), and (iii) a write-optimal local SkipList (denoted as `Local`), which manages a SkipList per NUMA node. `BR` and `Obl` manage one large PMTable, whereas `Deleg` and `Local` create four smaller PMTables. `Deleg` partitions key-value records according to hash keys, but `Local` allows a write client to insert data into the SkipList on its local NUMA node regardless of the key. Consequently, a read query has to search all four SkipLists. Even if a key is found in the local index, it must search remote indexes because a remote index may have a more recent update. Therefore, when there are n NUMA nodes, the ratio of local accesses is always $1/n$.

Our experiments, shown in Figures 11 and 12, run YCSB Load A (100 million inserts, 5-25 bytes string keys, 100-byte values) and Workload C (10 million queries).

PUT: Figure 11(b) shows that `Local` has the lowest write response time because it always inserts into the local PMTable. This eliminates remote NUMA node access for writes as shown in Figure 11(a). Braided SkipList (denoted as `BR`) has a higher write response time than `Local` because `BR` accesses remote NVMM via bottom layer pointers. Figure 11(a) shows that most NVMM accesses using `BR` are local, unlike NUMA-

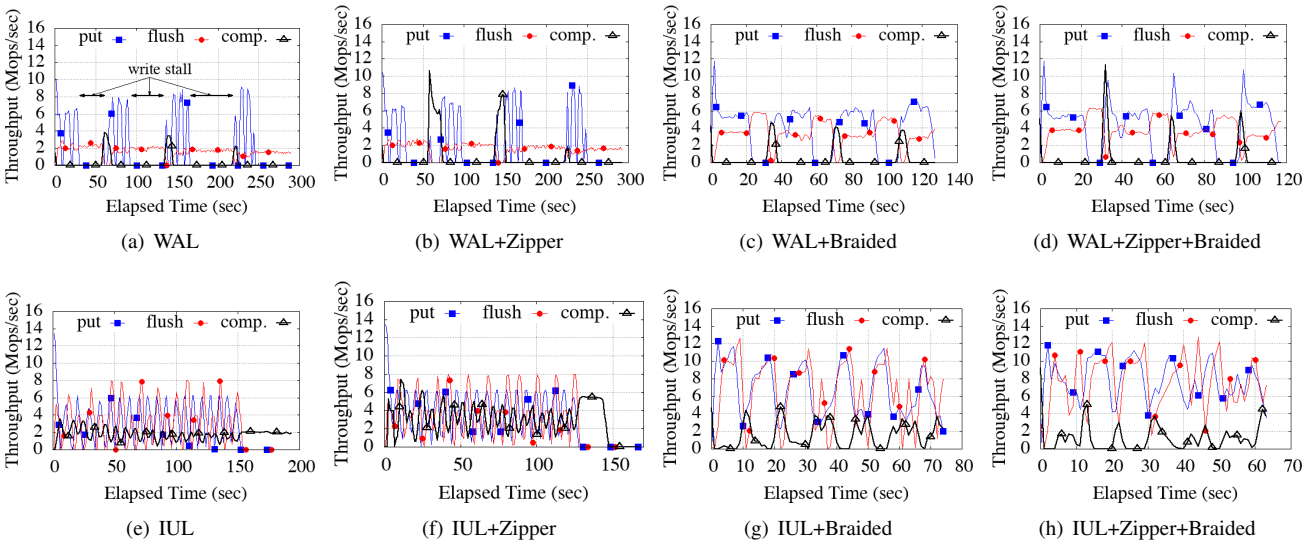


Figure 13: Put/Flush/Compaction Throughput over Time (YCSB Load A)

oblivious SkipList (20.1% vs. 74.1% remote accesses). `Obl` and `BR` access NVMM more than `Deleg` and `Local`.

Similar to `Local`, `Deleg` also completely removes remote NVMM access, but the write response time is significantly higher due to delegation overhead. That is, threads use slow atomic instructions to access the shared queue and make a memory copy for queries and results. Figure 11(b) shows that the queueing delay accounts for 77.1% of query response time with 80 client threads. Because put/get operations on a lock-free index are very lightweight, the synchronization overhead incurred by delegation dominates the overall response time.

GET: Figure 12(a) shows that the response time of `BR` for read queries is lower than the other methods. While `Local` outperforms `BR` for writes, the read response time of `Local` is about 4x higher than `BR` because `Local` must search all 4 PMTables. Although `BR` avoids visiting a more efficient search path that follows remote elements, Figures 11(a) and 12(a) show that it has almost no effect on the traversal length. `Deleg` shows the fewest memory accesses. However, due to synchronization overhead, its query response time is about 2x higher than `BR`, so its performance is even lower than `Obl`.

4.4 Putting It All Together

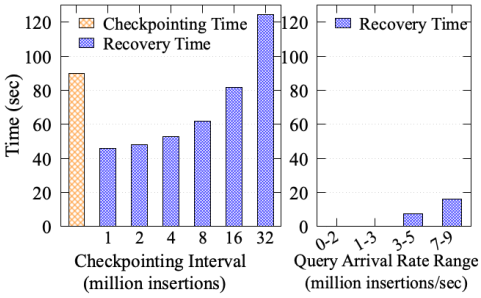
Figure 13 presents a factor analysis for ListDB. [4] We enable and disable each design feature of ListDB and measure write throughput (denoted `put`), flush throughput (MemTable $\rightarrow$ $L0$ PMTable, denoted `flush`), and compaction throughput ($L0 \rightarrow L1$ PMTable, denoted `comp.`) over time. We run 80 client threads and 40 background compaction threads for YCSB Load A, inserting 500 million 8-byte keys and 8-byte values. Figure 13(a) shows that disabling all three optimizations causes client threads to stall for more than 50 seconds. Enabling Zipper compaction improves the $L0 \rightarrow L1$ compaction throughput as shown in Figure 13(b), but the write stall problem still occurs because of the memory copy overhead for flushing the MemTable. If Braided SkipList is used, accessing remote NUMA nodes can be avoided when flushing the MemTable. Therefore, flush throughput doubles, which results in less frequent write stalls, as shown in Figure 13(c). Enabling both Zipper compaction and Braided SkipList results in shorter write stall times, and the workload completes in less than 120 seconds (Figure 13(d))

If IUL is used instead of WAL, `flush` throughput becomes comparable to `put` throughput, as shown in Figure 13(e). By avoiding expensive memory copy, write stalls are less frequent than WAL. However, note that compaction throughput is much lower than flush throughput. This increases the number of $L0$ PMTables and degrades search performance. As shown in Figure 13(f), if additionally IUL and Zipper compaction are enabled, the NVMM bandwidth improves by reducing the number of memory copies. Thus, it improves compaction and flush throughput. Enabling IUL and Braided SkipList, as shown in Figure 13(g), avoids NUMA effects, which improves both compaction and flush throughput. Finally, with all three optimizations enabled, the workload completes in under 65 seconds with virtually no write stalls (Figure 13(h)) compared to 300 seconds in figure 13(a).

4.5 Recovery Performance

We evaluate the recovery performance of asynchronous incremental checkpointing for ListDB and periodic synchronous checkpointing. Using the Facebook benchmark, we populate a database with 100 million objects and measure the time to recover using a checkpoint and write-ahead log entries. Despite using the same workload, the recovery performance of synchronous checkpointing is affected by the checkpointing interval, whereas asynchronous checkpointing is only affected by the query arrival rate. This is because the number of the

[4] Note that the scale of the x axis differs between the subfigures.



(a) Synchronous Checkpointing (b) ListDB

Figure 14: Recovery Performance

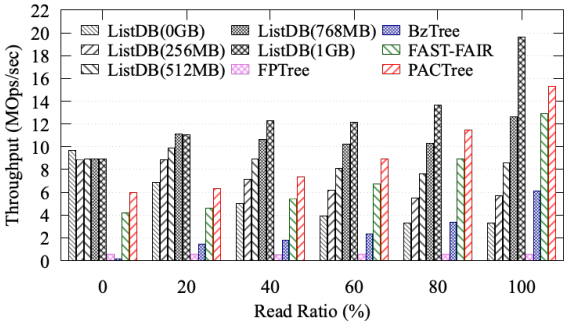


Figure 15: Comparison with Other Designs

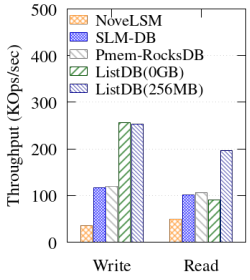


Figure 16: Comparison with NoveLSM and SLM-DB

log entries varies with asynchronous checkpointing, which background compaction threads have not yet merged into *L*1. If the query arrival rate is higher than the Zipper compaction throughput, the number of the IUL entries increases and the recovery process has to create a larger *L*0 PMTable with more log entries.

Figure 14 shows that a synchronous checkpointing takes about 90 seconds to serialize and flush the in-memory B+tree using the `binary_oarchive` class from the Boost library. This causes concurrent queries to block for 90 seconds while the checkpointing is being performed, resulting in unacceptably high tail latency. To alleviate the problem, checkpointing can be performed less frequently, but that increases the recovery time (i.e., the time to restore the checkpointed index and insert log entries to it) as more log entries accumulate.

In contrast, Figure 14 (b) shows that ListDB recovers instantly if it crashes when the write query arrival rate is lower than 3 million insertions/sec. If the query arrival rate varies between 7 and 9 million insertions/sec, ListDB takes about 19 seconds to recover. With a higher query arrival rate, the recovery time of ListDB increases.

4.6 Comparison with Other Designs

The experiments shown in Figure 15 compare the performance of ListDB with state-of-the-art persistent indexes; i.e., BzTree [1], FP-tree [49], FAST and FAIR B+tree [24], and PACTree [32], We run the experiments on a two-socket machine, because PACTree is hardcoded for two sockets. The two-socket machine has the same Intel Xeon Gold 5215 CPUs (40 logical cores in total), 128 GB DRAM (8x 16GB), and 1 TB (8x 128 GB) DCPMM. The database is pre-loaded with 100 million key-value records and then 40 clients submit 10 million queries with uniform distribution (generated from YCSB Workload A) with various read-write ratios. These tree-structured indexes are not optimized for (or do not support) large variable-length string keys and values. Therefore, we generated 8-byte numeric keys and 8-byte pointer values for the workload, which is favorable for tree-structured indexes with large fanouts.

Figure 15 shows that ListDB outperforms tree-structured persistent indexes for write-intensive workloads. For the write-only workload, `ListDB(0GB)` shows 79x, 17x, 2.3x and 1.6x higher throughput than BzTree, FPTree, FAST and FAIR B+tree, and PACTree, respectively. However, for the read-only workload, tree-structured indexes benefit from faster search performance. In particular, FAST and FAIR B+tree and PACTree show 3.88x and 4.61x higher search throughputs, respectively, than `ListDB(0GB)`. With the lookup cache enabled, ListDB outperforms or shows comparable performance to tree-structured indexes. The numbers in parentheses in the graph key show the lookup cache size. With a lookup cache larger than 768 MB, ListDB outperforms PACTree unless the read ratio is higher than 80%.

These results confirm that standard caching techniques can easily improve read performance. However, the lookup cache that indexes the location of key-value records cannot be used for PACTree, FAST FAIR B+tree, FPTree, etc. because they frequently relocate key-value records to different tree nodes due to tree rebalancing operations. That is, employing a DRAM cache for a tree-structured persistent index is not as simple as our address-only lookup caching. For example, Nap [57] has a very complicated caching mechanism.

4.6.1 Write Amplification

Although LSM trees have better write performance than tree-structured indexes, they have higher write amplification, as a critical limitation in block device storage [21, 41, 53]. To compare write amplification, we used Intel PMwatch [25] to measure the total number of accessed bytes in the experiments shown in Figure 15. All indexing methods suffer from high write amplification. DCPMM's internal write combining buffer transforms a small write (8-byte key and 8-byte value) into a 256-byte read-modify-write operation, resulting in at least 16x write amplification. In ListDB, the writes are further amplified by merge-sort operations in *L*0 and *L*1 PMTables. However, the write amplification of ListDB (104.4) is lower than that of FAST and FAIR B+tree (126.789) and comparable to that of PACTree (91.5) because ListDB merge-sorts SkipLists in-place.

4.7 Comparison with NoveLSM and SLM-DB

Figure 16 shows the single-threaded read and write throughput of NoveLSM, SLM-DB, Pmem-RocksDB, and ListDB.

The experiments run a single client thread (`db_bench`, 100 million random 8-byte keys and 1 KB values) because NoveLSM crashes when multiple threads concurrently access the database. NoveLSM and SLM-DB were designed to use NVMM as an intermediate layer on top of the block device file system, but our experiments store all SSTables in NVMM formatted with EXT4-DAX for a fair comparison.

NoveLSM shows the worst performance, not because of its design but because it is implemented on top of LevelDB, which is known to have poor performance. SLM-DB is also implemented on top of LevelDB but shows better performance because it uses FAST and FAIR B+tree as its core index. Since SLM-DB is not yet ported to use PMDK, it has no overhead imposed by run-time flushing or transactional updates, i.e., it shows DRAM performance and does not survive a system crash. Nonetheless, SLM-DB does not show better performance than Pmem-RocksDB, a fully persistent key-value store. Compared to Pmem-RocksDB, `ListDB(0GB)` shows twice the write throughput, but read performance is slightly worse unless the lookup cache is enabled. This is because Pmem-RocksDB benefits from memory locality by storing keys contiguously in NVMM in sorted order, whereas ListDB does not relocate data.

4.8 Comparison with Pmem-RocksDB

Finally, we compare the performance of ListDB with Intel's Pmem-RocksDB using the *Prefix Dist* workload in the Facebook benchmark. The experiments shown in Figure 17 run 80 client threads and use the default key and value sizes of the benchmark (48-byte string keys and variable-length values ranging from 16 bytes to 10 KB). The workload submits queries according to a query arrival rate (QPS parameter) that follows a sine distribution with a noise factor of 0.5. The put/get ratio of the workload is 3 to 7.

For various parameter settings, ListDB consistently outperforms Pmem-RocksDB. The results of two different settings are presented in Figure 17 - an `idle` workload (0.1 $\sim$ 0.3 million write queries and 0.2 $\sim$ 0.7 million read queries arrive per second; 200 million queries in total), in which the throughput of Pmem-RocksDB is saturated, and a `heavy` workload (2.4 $\sim$ 7.2 million write queries and 5.6 $\sim$ 16.8 million queries arrive per second; 5 billion queries in total), in which the throughput of ListDB is saturated. The lookup cache is disabled for ListDB while setting the maximum DRAM usage for both key-value stores to 1 GB and allowing Pmem-RocksDB to use the default 8 MB block cache.

For the `idle` workload, Pmem-RocksDB suffers from excessive NVMM writes, so its `put` throughput saturates at 200 Kops. For the Facebook benchmark, a `get` query has to wait for its previous `put` query to commit. Therefore, the `get` throughput of Pmem-RocksDB saturates at 400 Kops in the experiment. In contrast, Figure 17(b) shows that the throughput of ListDB follows the sine distribution, i.e., the query arrival rate, without blocking queries.

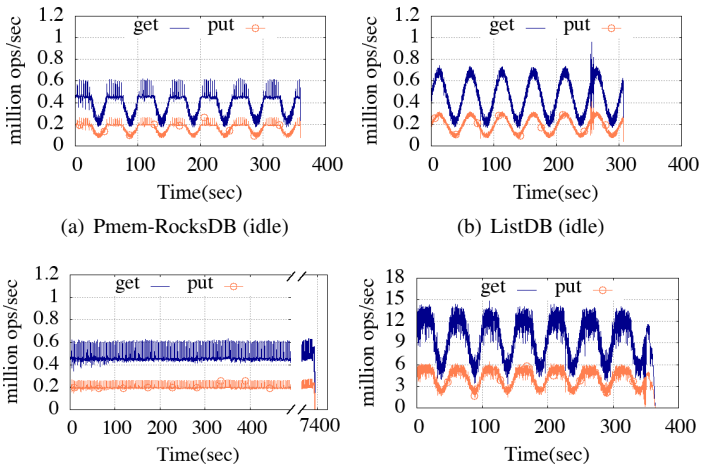


Figure 17: Throughput over Time (Facebook Benchmark)

For the `heavy` workload, Pmem-RocksDB's throughput is still saturated. On the other hand, the `put` throughput of ListDB is 25x higher than that of Pmem-RocksDB, i.e., 5 million ops. Similarly, the `get` throughput of ListDB is up to 22x higher than that of Pmem-RocksDB (i.e., 13 million vs. 0.6 million ops). As such, ListDB completes the workload 19.4x faster than Pmem-RocksDB (i.e., 380 vs. 7400 seconds).

5 Conclusion

In this work, we design and implement ListDB - a key-value store that leverages the byte-addressability to avoid data copies by restructuring data in-place and high-performance of NVMM to reduce the write amplification and avoid write stalls. We show that ListDB significantly improves write performance via asynchronous incremental checkpointing and in-place compaction. With its three-level structure, ListDB outperforms state-of-the-art persistent indexes and NVMM-based key-value stores in terms of write throughput. A standard lookup cache can help mitigate the problem of having multiple levels. For future work, we are exploring the possibility of improving search performance by introducing another level, namely L2 PMTable, to opportunistically rearrange L1 PMTable elements for spatial locality and garbage collection.

Acknowledgement

We would like to give special thanks to our shepherd Dr. Marc Shapiro and to the anonymous reviewers for their valuable comments and suggestions. This research was supported in part by Samsung Electronics, and also by the R&D program of National Research Foundation of Korea (NRF) (grant No. NRF2018R1A2B3006681) and IITP (grant No. 2018-0-00549 and 2021-0-01817) and Electronics and Telecommunications Research Institute (ETRI) grant (grant No. 20ZS1310) funded by the Korean government. The corresponding author is Beomseok Nam.

References

[1] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, jan 2018.

[2] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache storage for the next billion. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 123–136, 2009.

[3] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 363–375, 2017.

[4] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[5] Trevor Brown. Reclaiming Memory for Lock-Free Data Structures: There has to be Better Way. In *Proceedings of the 34th ACM Symposium on the Principles of Distributed Computing (PODC'15)*, 2015.

[6] Irina Calciu, Justin Gottschlich, and Maurice Herlihy. Using Elimination and Delegation to Implement a Scalable NUMA-Friendly Stack. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism (HotPar 13)*, June 2013.

[7] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-Box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 207–221, 2017.

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[9] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 1007–1019, 2018.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. BigTable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[11] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[12] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 1077–1091, 2020.

[13] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free Concurrent Level Hashing for Persistent Memory. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 799–812, July 2020.

[14] Nachshon Cohen and Erez Petrank. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures (SPAA'15)*, 2015.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.

[16] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, page 339–351, 2021.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*.

[18] Facebook. db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[19] Thomas E. Harta, Paul E. McKenneyb, Angela Demke Brown, and Jonathan Walpole. Performance of Memory

Reclamation for Lockless Synchronization. *Journal of Parallel and Distributed Computing*, 67:1270–1285, 2007.

[20] HBase. https://hbase.apache.org/.

[21] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 127–144, 2017.

[22] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Simple Optimistic Skiplist Algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity (SIROCCO)*, pages 124–138, 06 2007.

[23] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[24] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 187–200, 2018.

[25] Intel. PMWatch. https://github.com/intel/intel-pmwatch.

[26] Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[27] Varun Jain, James Lennon, and Harshita Gupta. LSM-Trees and B-Trees: The Best of Both Worlds. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, page 1829–1831, 2019.

[28] Ashok Joshi, William Bridge, Juan Loaiza, and Tirthankar Lahiri. Checkpointing in Oracle. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, page 665–668, 1998.

[29] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, pages 191–205, 2019.

[30] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 993–1005, 2018.

[31] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. BoLT: Barrier-Optimized LSM-Tree. In *Proceedings of the 21st International Middleware Conference (Middleware)*, page 119–133, 2020.

[32] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, page 424–439, 2021.

[33] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[34] D. Lea. Java Platform SE 8, java.util.concurrent.ConcurrentSkipListMap. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html.

[35] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 257–270, 2017.

[36] LevelDB. https://github.com/google/leveldb.

[37] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–13, 2011.

[38] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees: Optimizing Persistent Index Performance on 3DX-Point Memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, mar 2020.

[39] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, 2018.

[40] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment*, 13(8):1147–1161, apr 2020.

[41] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 133–148, 2016.

[42] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 207–219, 2015.

[43] Fei Mei, Qiang Cao, Hong Jiang, and Lei Tian Tintri. LSM-tree Managed Storage for Large-scale Key-value Store. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, pages 142–156, 2017.

[44] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. CPHASH: A Cache-Partitioned Hash Table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 319–320, 2012.

[45] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage (FAST)*, pages 31–44, 2019.

[46] Suman Nath and Aman Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 410–419, 2007.

[47] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[48] Oracle Berkeley DB. https://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[49] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 371–386, 2016.

[50] Persistent Memory Development Kit (PMDK). https://pmem.io/pmdk/.

[51] PMEM-RocksDB. https://github.com/pmem/pmem-rocksdb.

[52] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[53] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 497–514, 2017.

[54] RocksDB. https://rocksdb.org/.

[55] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment*, 14(11):2216–2229, jul 2021.

[56] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[57] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 93–111, July 2021.

[58] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 1996.

[59] Paul R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. *SIGARCH Computeer Architecture News*, 19(4):6–13, jul 1991.

[60] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 349–362, 2017.

[61] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, February 2016.

[62] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, pages 169–182, February 2020.

[63] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai Leong Yong. NV-Tree: Reducing Consistency Const for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX conference on File and Storage Technologies (FAST)*, 2015.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

OdinFS: Scaling PM Performance with Opportunistic Delegation

Diyu Zhou *Yuchen Qian* *Vishal Gupta* *Zhifei Yang* *Changwoo Min*[†] *Sanidhya Kashyap*

EPFL [†]Virginia Tech

Abstract

Existing file systems for persistent memory (PM) exploit its byte-addressable non-volatile access with low latency and high bandwidth. However, they do not utilize two unique PM properties effectively. The first one is contention awareness, *i.e.*, a small number of threads cannot thoroughly saturate the PM bandwidth, while many concurrent accesses lead to significant PM performance degradation. The second one is NUMA awareness, *i.e.*, exploiting the remote PM efficiently, as accessing remote PM naively leads to significant performance degradation.

We present OdinFS, a NUMA-aware scalable datapath PM file system that addresses these two challenges using a novel ***opportunistic delegation*** scheme. Under this scheme, OdinFS decouples the PM accesses from application threads with the help of background threads that access PM on behalf of the application. Because of PM access decoupling, OdinFS automatically parallelizes the access to PM across NUMA nodes in a controlled and localized manner. Our evaluation shows that OdinFS outperforms existing PM file systems up to 32.7× on real-world workloads.

1 Introduction

Persistent memory (PM), a storage-class memory, breaks the traditional dichotomy of storage and memory. It offers byte addressability, non-volatility, low latency, and high bandwidth [8, 14, 23, 43]. Recent characterization studies show that PM has many subtle performance characteristics [18–20, 23, 27, 29, 37, 39, 40, 43], posing a significant challenge for storage stacks to utilize PM performance efficiently.

Such a challenge arises from two unique PM characteristics. The first factor is the tension between concurrent accesses and PM performance. In particular, a small number of threads underutilize PM bandwidth, while a high number of concurrent access threads[1] lead to PM performance meltdown [39]. The meltdown happens because a high number of concurrent access threads render the caching and prefetching in PM inefficient [14, 43]. The second factor is the pronounced NUMA impact on PM, as several prior works found that remote NUMA accesses on PM are much slower than DRAM, leading to at least 2× bandwidth reduction [14, 27, 38]. Supporting multiple NUMA domains is currently the primary way to increase PM's capacity and aggregated bandwidth. Unfortunately, the pronounced NUMA impact defeats the purpose of PM NUMA architecture.

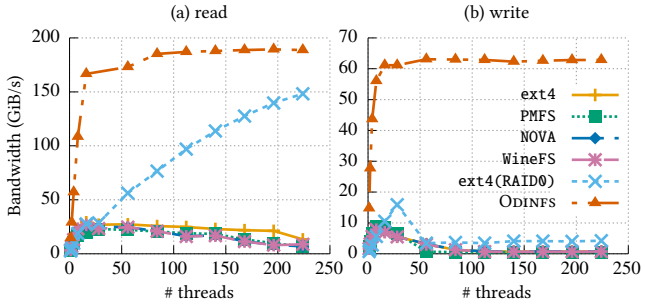


Figure 1: The maximal PM read and write bandwidth of four PM file systems and OdinFS. `ext4(RAID0)`: `ext4` mounted on a RAID0 built from PM across all NUMA nodes. Benchmark: `fio`, in which each thread accesses a private file at the granularity of 2MB sequentially, on an eight-socket machine.

Several proposed PM file systems exploit various characteristics of PM [10, 12, 16, 17, 24, 25, 28, 34, 41]. However, none of the existing PM file systems considers the tension between concurrent accesses and PM performance. Moreover, the conventional approach to mitigate the PM NUMA impact is to migrate data to CPUs or vice versa [25, 42], which incurs a high migration overhead, and cannot efficiently utilize the aggregated PM bandwidth. Figure 1 illustrates the issues for several PM file systems [2, 17, 25, 41]. The bandwidth of these file systems highly depends on the thread counts. Except for `ext4(RAID0)` performing the read workload, the read and write bandwidth of these file systems collapse after the thread count exceeds a certain threshold. In summary, existing PM file systems cannot efficiently utilize PM in a NUMA setup and incur performance collapse if multiple threads of

[1] In this paper, we use the term "access thread" to denote a thread that directly accesses PM. It could be an application thread or a kernel thread.

the application (*e.g.*, file server software and video streaming software) access PM concurrently.

This paper presents OdinFS: (Opportunistic DelegatIoN File System), a NUMA-aware PM file system that maximizes PM performance by controlling concurrent accesses, minimizing the NUMA impact, and parallelizing PM accesses to utilize the aggregated bandwidth. To design OdinFS, we first holistically analyze the behavior of existing PM file systems on an eight-socket NUMA machine. We analyze two issues specifically: maintaining maximum PM performance within a NUMA node and minimizing the PM NUMA impact. For the first issue, we find that both read and write performance of PM collapse with high thread counts, while prior work only reports the write performance collapse [14, 43]. For the second issue, we provide a detailed analysis and quantitatively confirm that placing the access threads in the same NUMA node as PM minimizes the NUMA impact.

Motivated by our performance analysis, we design OdinFS with three major design goals: (1) **Limit concurrent PM accesses (access arbitration)**: OdinFS controls the number of PM access threads to maintain the maximal PM performance within a NUMA node. (2) **Localized PM accesses (NUMA-awareness)**: OdinFS ensures threads always access the local PM within a NUMA node, thereby avoiding the PM NUMA impact. (3) **Automatic parallel PM accesses (automatic parallelization)**: OdinFS automatically parallelizes applications' PM access requests across all NUMA nodes without application modification. OdinFS thus efficiently utilizes aggregated PM bandwidth, thereby improving application performance.

OdinFS achieves these goals by proposing a new approach—***opportunistic delegation***—that decouples PM data accesses from application threads. Specifically, on each NUMA node, OdinFS creates multiple background kernel threads (delegation threads) that access PM on behalf of the application threads. Therefore, OdinFS limits the maximum concurrency within PM by controlling the number of delegation threads. Moreover, the delegation threads are local to each NUMA node, leading to NUMA-aware localized accesses. OdinFS thus departs from the conventional NUMA-mitigation approaches in PM file systems that mainly involve data or thread migration.

Furthermore, the delegation threads enable servicing bulk data requests by efficiently utilizing aggregated PM bandwidth across all NUMA nodes. Specifically, OdinFS first stripes the file data across PM in all NUMA nodes. Exploiting the well-designed system call interface, OdinFS services data system calls (*e.g.*, `read()` and `write()`) by transparently dividing them into multiple disjoint access requests based on the stripe size and sending such access requests to the corresponding delegation threads. The delegation threads thus access PM in different NUMA nodes in parallel to serve the system call. We further enhance OdinFS with fine-grained parallelism for data operations. Our evaluation shows that OdinFS outperforms other file systems by up to 32.7× for real-world workloads and has up to two orders of magnitude performance improvement for scalability microbenchmarks.

This paper makes the following contributions:

- **Analysis.** We thoroughly analyze the behavior of existing PM file systems on a large NUMA machine and reveal two new findings.
- **Opportunistic delegation.** We propose an opportunistic delegation scheme for PM file systems that decouples PM data accesses from application threads, thus efficiently utilizing both local and remote PM.
- **OdinFS** We design and implement OdinFS: a PM file system that builds on the opportunistic delegation scheme with state-of-the-art concurrency control mechanisms. OdinFS maximizes the performance and further scales data operations with increasing threads.

2 PM Performance Analysis

Prior study has shown that the underlying architecture of PM is quite complicated [39], and PM has limited bandwidth and higher latency compared to DRAM [14, 40]. Moreover, naively utilizing PM in NUMA machines often underutilizes PM or leads to performance collapse [14, 25, 27, 38]. To showcase the issues of concurrent NUMA accesses in PM, we first provide an overview of the current hardware (§2.1). We then analyze why existing PM file systems fail to handle many concurrent requests (§2.2) and the impact of different types of accesses in a NUMA machine (§2.3).

2.1 Intel Optane internals

The Intel Optane [8] PM is the only publicly available nonvolatile memory technology so far. A memory controller accesses PM at the granularity of a cache line (64 bytes). However, the access size of the internal 3D-Xpoint storage media is 256 bytes. Such an access size mismatch results in read or write amplification. The 3D-Xpoint media includes a buffer (*XPBuffer*) and an associated prefetcher (*XPPrefetcher*) to address this issue and mitigate its long latency. The XPBuffer combines adjacent accesses to PM, and the XPPrefetcher prefetches blocks in the 3D-Xpoint media to XPBuffer based on the access pattern. In addition, the 3D-Xpoint media also stores the inter-NUMA node coherence information [6, 27]. Hence, inter-NUMA accesses may involve writing to the 3D-Xpoint media to update the coherence information, which is the root cause of the slow inter-NUMA PM accesses (§2.3).

2.2 Concurrent accesses to PM

Prior work [14, 43] finds that PM performance depends on the access size and the number of concurrent access threads. The impact of access size is well understood. Applications accessing PM perform well as long as the access size is large enough to stress all the interleaved PM DIMMs. To understand the impact of concurrent access threads, we run `fio` [4], in which each pinned thread sequentially accesses a private 1GB file at a 2MB granularity. We evaluate on an eight-socket

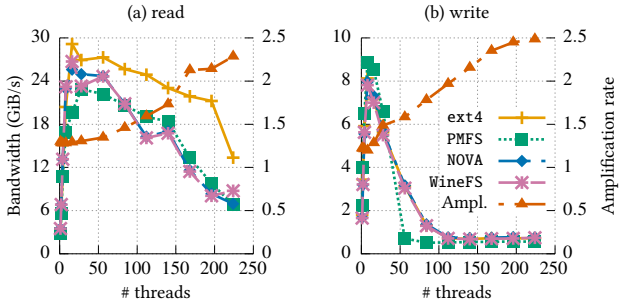


Figure 2: PM read and write bandwidth of PM file systems with increasing threads for sequential 2MB access size. Results show that both read and write performance collapse after exceeding a specific limit. We observe a dramatic increase in read/write amplification due to cache thrashing in the PM storage device.

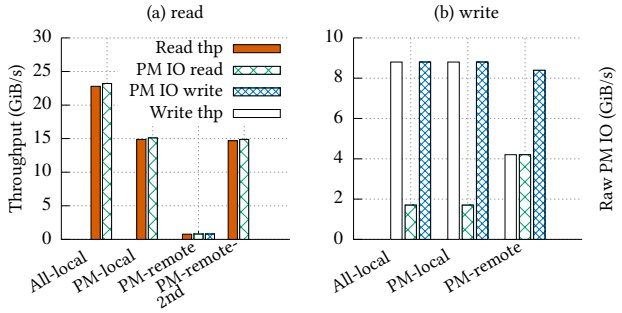


Figure 3: Application throughput and the raw PM I/O for reading data from PM (left) and writing data to PM (right) with the same workload in §3.6 and the following configurations. *All-local*: Access threads, PM, and DRAM (I/O buffer) are in NUMA node 0. *PM-local*: Access threads and PM are in NUMA node 0; DRAM is in NUMA node 1. *PM-remote*: PM is in NUMA node 0; access threads and DRAM are in NUMA node 1. *PM-remote-2nd*: A consecutive run with *PM-remote*.

NUMA machine, with each socket having six interleaved Optane DIMMs and a processor with 28 cores.

Figure 2 shows the read and write bandwidth of PM file systems with increasing threads. Both read and write reach their peak bandwidth with sixteen and eight threads, respectively. After that, increasing threads severely degrades the overall bandwidth. Specifically, with 224 threads, the read and write bandwidth degrades by 3.7× and 17.2× compared to the peak bandwidth, respectively.

Such performance collapse occurs because high concurrent accesses thrash the underlying cache of PM.[2] Specifically, a mismatch exists between the CPU access size (64 bytes) and the underlying PM storage access size (256 bytes). PM minimizes the read/write amplification overhead by batching writes (XPBuffer) and prefetching (XPPrefetcher) (§2.1). However, with high concurrent accesses, the sequential accesses from different threads convert into non-adjacent accesses. These accesses arrive at the PM simultaneously, which reduces the efficiency of both caching and prefetching. As a result, it increases read and write amplification, as XPBuffer cannot keep up with the requests and the underlying PM media latency starts to dominate for fetching or writing data, leading to performance collapse. The read collapse threshold is higher than write since reads perform better than writes with the 3D-Xpoint media. Thus, PM can sustain the read bandwidth despite reducing caching and prefetching efficiency up to two NUMA nodes.

We find that both read and write bandwidth crashes after a certain point. The results for writes are consistent with prior works [14, 43]. However, we find that read bandwidth also starts to collapse after two NUMA nodes. This is contrary to prior work, which reports that the read bandwidth scales with increasing threads.

Insight #1. A file system must control the number of threads that concurrently access PM for both reads and writes to preserve the maximal performance within a NUMA node.

[2]We verify that the performance collapse is not due to the scalability bottleneck in the file systems by confirming that most of the CPU cycles are spent in accessing PM.

2.3 NUMA impact on PM

We now analyze the NUMA effect on PM. `WineFS` [25] proposes to migrate a thread to a PM NUMA node to mitigate the NUMA impact [25]. Unfortunately, there is no in-depth analysis of the effectiveness of this mechanism. Specifically, suppose a thread copies data between remote PM and local DRAM. In this case, migrating the thread to the respective PM NUMA node still involves remote memory access, and thus it is not clear why or how the thread migration can improve performance.

We answer the aforementioned question by investigating the performance impact of NUMA placements of thread, DRAM, and PM. We configure `fio` with three setups. In the *All-local* setup, threads, PM, and DRAM (*i.e.*, I/O buffer) are in the same NUMA node (node 0), which serves as the best-case scenario. In the *PM-local* setup, threads and PM are in the same NUMA node (node 0), while DRAM is in a remote NUMA node (node 1). In the *PM-remote* setup, PM is in NUMA node 0, while threads and DRAM are in the same NUMA node (node 1). We evaluate this experiment using `PMFS` with 28 threads for read and 8 threads for write. Other file systems or thread counts produce similar results.

Figure 3 shows the PM read and write throughput with the three setups. We note that both *PM-local* and *PM-remote* perform the same task of copying data between PM on NUMA node 0 and DRAM on NUMA node 1. However, *PM-local* achieves nearly 19.1× and 2.1× higher throughput than *PM-remote*. Furthermore, *PM-local* achieves 60% and almost 100% of *All-local* read and write throughput, respectively. The above results are due to the implementation of the *directory coherence protocol* in Intel machines [6, 27]. Specifically, Intel maintains intra-NUMA and inter-NUMA directory information separately [5]. The processor cache and memory store the intra-NUMA directory and inter-NUMA directory information, respectively. With the *PM-local* setup, the PM cache blocks become NUMA-local: data is written to the processor

cache; while the DRAM cache block moves between NUMA nodes. Hence, the system updates the PM directory locally, while writing to DRAM to update its directory information. However, with the *PM-remote* setup, the processor updates DRAM directory information on the processor cache, while updating its directory information on PM. Hence, the performance difference between DRAM and PM leads to the performance difference between *PM-local* and *PM-remote*.

To verify the above claim, we used Intel PCM [7] to measure the PM device level read/write IO, as shown in Figure 3. For *All-local* and *PM-local*, the total bytes written and read from the PM device match the actual I/O bandwidth, indicating no directory information update. However, *PM-remote* incurs extra read and write traffic to the PM device. Furthermore, a consecutive read with *PM-remote* (*PM-remote-2nd*) can restore the performance, while a consecutive write in *PM-remote* still suffers from NUMA impact. The above evidence confirms that *PM-remote* involves directory coherence updates. Specifically, the extra read and write traffic is due to updating the coherence information. The *PM-remote-2nd* setup can only restore the read performance since, in this case, coherence information update is not needed for read but is still required for write. In summary, our performance analysis quantitatively confirms that placing the access threads and PM in the same NUMA node minimizes the NUMA impact on PM.

Insight #2. To minimize the pronounced PM NUMA impact, and efficiently utilize remote PM, a file system should place the access threads local to the PM.

3 OdinFS Design

Following our performance analysis on PM (§2), we present OdinFS, a NUMA-aware PM file system that maximizes PM performance within and across NUMA nodes through opportunistic delegation. This section first presents the design goals that enable OdinFS to maximize PM performance (§3.1), an overview of OdinFS (§3.2), followed by the design of each individual component.

3.1 OdinFS Design Goals

We design OdinFS to meet the following goals:

- **Limiting concurrent PM access (access arbitration).** To avoid the PM performance collapse with many concurrent PM accesses (§2.2), OdinFS should limit the concurrency to maximize PM performance within a single NUMA node.
- **Localized PM access (NUMA-awareness).** To avoid the performance collapse due to the PM NUMA impact, OdinFS only allows threads to access local PM (§2.3). This minimizes the PM NUMA impact and opens the opportunity for OdinFS to utilize remote PM efficiently.
- **Automatic parallel PM access (automatic parallelization).** The access arbitration and NUMA-aware design goals allow OdinFS to maximize the local and remote PM performance. To fully benefit from the aggregated PM bandwidth, OdinFS further parallelizes large PM accesses across all NUMA nodes automatically without application changes.
- **Scalability.** Scalability is the overarching goal of OdinFS. Modern machines have multiple NUMA nodes and hundreds of CPUs. The above three design goals allow OdinFS to scale PM performance with an increasing core count. Beyond that, OdinFS should maximize concurrent accesses within the same file.

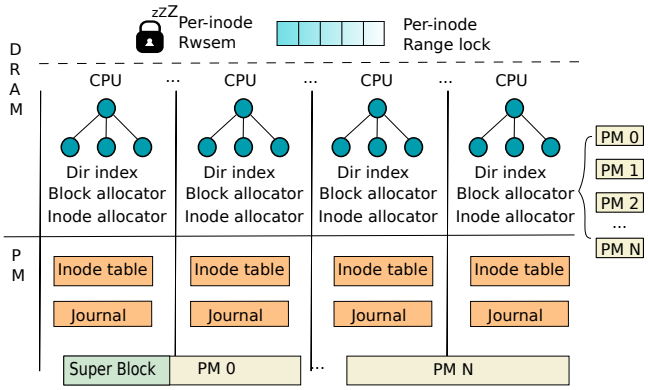


Figure 4: OdinFS architecture. OdinFS maintains per-CPU data structures to minimize the synchronization overhead. Furthermore, OdinFS maintains the directory index, block allocator, and inode allocator in DRAM to maximize performance. OdinFS enhances the per-inode readers-writer semaphore with our optimized range lock to increase concurrency.

3.2 OdinFS Architecture

Figure 5 shows the key components of OdinFS and their typical workflow. We next present the key design of OdinFS and explain how they meet the design goals of OdinFS (§3.1).

(1) NUMA-striped data layout for cumulative PM bandwidth utilization. Unlike other NUMA-aware PM file systems that try to localize file accesses within a single PM NUMA node [25, 42], OdinFS stripes the data of every file across PM on each NUMA node in a round-robin manner. OdinFS makes this design choice since it can minimize the PM NUMA impact *with delegation*, as detailed below. Furthermore, stripping file data across PM enables OdinFS to exploit all available PM bandwidth to handle application requests, which opens the door for automatic request parallelization.

(2) Delegation-based PM accesses to maximize PM performance. A key insight in OdinFS is that the access arbitration, NUMA-awareness, and automatic parallelization design goals can be simultaneously achieved by decoupling PM data accesses from application threads through delegation. In particular, for each NUMA node, OdinFS creates several background threads (delegation threads). Only the delegation threads can access PM. When the application thread needs to access PM, it first checks which NUMA node the PM address belongs to and then sends the PM access requests to one of the delegation threads on that NUMA node. The delegation thread performs the access on behalf of the

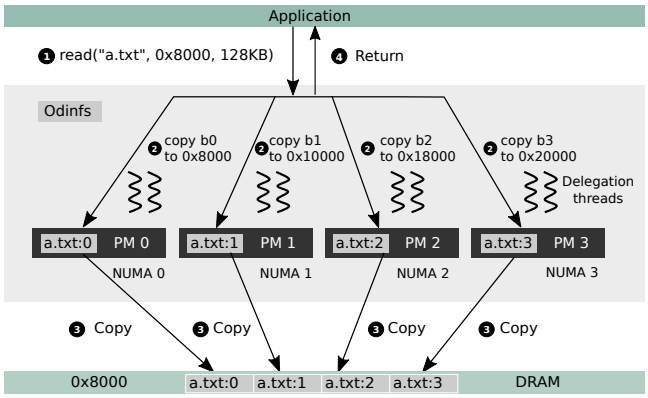


Figure 5: Overview of Odinfs. Each NUMA node has delegation threads that access local PM on behalf of application threads. Odinfs stripes the file data across all PM NUMA nodes. ❶ An application thread issues a read system call. ❷ Odinfs divides the system call into multiple access requests based on the stripe size and sends them to the delegation threads. ❸ The delegation threads read from PM in different NUMA nodes in parallel to service the access requests. ❹ The application thread returns.

application thread and informs the application thread when the access completes.

Since only the delegation threads can access PM, they effectively act as a central entity to *arbitrate* PM access. Regardless of the application thread count, delegation threads decide the level of concurrent accesses to PM. Thus, Odinfs accesses PM with a thread count that avoids the PM performance collapse with many concurrent access threads (§2.2). This effectively achieves the *arbitration* design goal. Since the delegation threads are in the same NUMA node as PM, Odinfs always access PM locally, in either *All-local* or *PM-local* setup (§2.3). Thus, Odinfs minimizes the PM NUMA impact, achieving the *NUMA-aware* design goal.

(3) Automatic parallelization at the system call boundary. The data striping and the delegation threads allow Odinfs to serve IO requests from applications in parallel across all the NUMA nodes. Moreover, the POSIX interface enables Odinfs to *automatically* parallelize the requests without modifying applications. Specifically, Odinfs divides all data system call (*e.g.*, `read`, `write`, `pread`, `writev`) requests into multiple independent sub-requests based on the stripe size, and sends them to the corresponding delegation threads. The delegation threads then execute these requests by accessing PM in different NUMA nodes in parallel. Figure 5 illustrates the case. In this way, Odinfs achieves the *automatic parallelization* design goals.

(4) High scalability with full PM performance. Delegating PM access allows Odinfs to maximize PM performance. Odinfs further maximizes concurrent accesses to ensure applications can benefit from the performance gains even under the high contention case. Specifically, most existing PM file systems globally protect the inode [2, 17, 25, 41]. Odinfs further increases the disjoint data access parallelism with a readers-writer range lock for each inode. This enables concurrent writes to disjoint regions and concurrent reads from the same file region. The use of range lock poses a significant challenge for enforcing crash consistency. Odinfs overcomes this issue by preserving the whole inode lock and falling back to it for concurrency control if needed. (§3.7).

Odinfs design novelty. To the best of our knowledge, Odinfs is the first PM file system that addresses the goal of access arbitration and automatic parallelization. While some NUMA-aware file systems mitigate PM NUMA impact [25], they either move computation to data [25], or move data to computation [42]. Odinfs proposes a fundamentally different approach by using the delegation to minimize the NUMA impact. Furthermore, Odinfs extends the scope of the NUMA-aware file systems. Instead of focusing only on minimizing the NUMA impact, Odinfs takes a radical approach of parallelizing and striping data across all PM NUMA nodes for the best performance. Moreover, our controlled PM access approach also minimizes I/O amplification (§2.2), leading to low write amplification. Lower write amplification further increases the life of the PM device and minimizes the long latency that happens due to wear leveling [39].

3.3 Handling system calls

Odinfs is a POSIX-compliant in-kernel file system. A key novelty in Odinfs is the PM access delegation. However, since delegation incurs communication overhead, Odinfs does not delegate small PM accesses (§3.5). As a result, the delegation threads only perform data operations, while the metadata operations (*e.g.*, `open`, `close`) are handled by application threads. Furthermore, application threads directly access PM for small data operations.

Handling bulk data operations with delegation. After issuing a data system call (*e.g.*, `write`) and entering into the kernel space, the application thread divides the requests into multiple sub-requests, each consisting of a data stripe (§3.4). For each access request, the application thread walks the indexing structure and obtains the corresponding address on PM and the NUMA domain. It then enqueues the request (*e.g.*, source and destination memory address, access length) on a corresponding ring buffer of the PM NUMA node. After enqueueing all requests, the application thread waits for delegation threads to complete and then returns to the user space. Meanwhile, a delegation thread receives the request via the ring buffer. The delegation thread dequeues the request and accesses PM on behalf of the application thread by copying the data between PM and the specified DRAM buffer. After completing the request, the delegation threads notify the application threads.

3.4 NUMA-aware PM allocation

NUMA-aware allocator. To operate on multiple PM NUMA nodes and stripe file data across them (§3.2), Odinfs designs a NUMA-aware PM allocator. Odinfs inherits the performant and scalable allocator design from `NOVA` and `WineFS` and ex-

tends the allocator design to handle multiple PM NUMA nodes. OdinFS uses a per-CPU allocator residing on DRAM, in which each CPU owns multiple private PM pools, each corresponding to one PM NUMA node (Figure 4). Block allocation works as follows: The allocator receives the allocation request with a block size and a NUMA node as arguments. It first tries to serve the request from its own per-CPU PM pool. If that fails, it tries to serve the requests from the pool in the specified NUMA node having the largest free space. If both attempts fail, the allocator returns an error.

Layout policy. OdinFS employs different layout policies for file data, indexing structure, and other metadata. OdinFS stripes the file data across all PM NUMA nodes to enable parallel access (§3.2). Since OdinFS does not delegate small PM accesses (§3.5), it optimizes for small files by placing the first stripe of each file in the local PM node as the creation thread, if possible. This assumes that the following accesses are likely from the same NUMA node thanks to temporal locality. OdinFS places the remaining stripes in a round-robin fashion across all PM NUMA nodes. With our system, the memory controller in each NUMA node interleaves six PM DIMMs at 4KB granularity. Therefore, we set the stripe size as 32KB to maximize PM performance. Since OdinFS does not delegate access to the indexing structure, OdinFS places it in PM local to the CPU that creates the file, leveraging temporal locality. Regarding other metadata, OdinFS places the superblock in the first PM NUMA node and per-CPU metadata (*e.g.*, journaling) in the local PM node (Figure 4).

3.5 Opportunistic delegation

Since delegating PM accesses involves communication overhead, it is not always beneficial, especially for small accesses. Thus, OdinFS performs opportunistic delegation only for PM accesses that might improve the performance. Based on our performance analysis (§2), OdinFS uses different delegation policies for PM reads and writes.

Write access. OdinFS always delegates write accesses with an access size larger than 256 bytes (XPBuffer size) to limit the performance collapse and minimize the NUMA impact.

Read access. Unlike writes, OdinFS chooses a more relaxed policy for delegating reads. Specifically, PM read performance starts to collapse with a high thread count (> 56). Furthermore, the PM read performance can be restored after repetitive remote reads (PM-remove vs. PM-remote-2nd in Figure 3). Thus, OdinFS checks the number of threads that read from each PM NUMA node for every fixed interval. If it finds that for one PM NUMA node, the number of threads is constantly higher than the collapse threshold (56), OdinFS arbitrates access to that PM device by using the same policy as write. Otherwise, OdinFS only delegates the read accesses that may benefit from the automatic parallelization (*i.e.*, those with access size larger than the stripe size: 32KB). We find this policy is enough to achieve good performance.

Saving CPU cycles. A delegation thread uses a variant of the spin-then-park strategy to 1) avoid wasting CPU cycles when there is no request and 2) minimize the long latency due to naively parking and waking up threads [26]. OdinFS uses the IO size of application threads as a heuristic to decide the length that a delegation thread should spin before parking. The spinning is inversely proportional to the IO request size. For example, for large IO requests, delegation threads spin for a shorter duration because they can amortize the parking/wake-up latency by handling long requests. On the other hand, delegation threads spin for a longer duration for small IO requests, since we assume that application threads are likely to issue sparse requests in this case. We find that this heuristic works well for every evaluated workload.

3.6 Concurrency control

Prior in-kernel PM file systems [17, 25, 41] rely on VFS's inode lock for concurrency control. Inspired by recent works [12, 36], OdinFS increases fine-grained access to a file with a per-inode readers-writer range lock. The lock allows parallel writes to disjoint regions, while concurrent reads on the same region in a file. The existing concurrency control mechanisms still protect other operations.

3.7 Crash consistency

Consistency mode. OdinFS currently supports two consistency modes: POSIX and sync [24]. The POSIX mode guarantees that all metadata operations are synchronous and atomic (*e.g.*, `ext4`). The sync mode is the default setup that in addition to POSIX mode, further ensures that all data operations are synchronous but not atomic. Specifically, when the system call returns, the data is guaranteed to persist on PM. However, a crash may cause data operations being partially completed. This provides the same guarantee as `PMFS` and the "relaxed mode" of `NOVA`. If required, we can extend OdinFS to provide other consistency modes [24, 25, 41].

Atomic updates and per-CPU journaling. OdinFS provides metadata crash consistency with atomic updates [17] and per-CPU journaling [41]. Intel architecture only supports 8 bytes as atomic updates and (aligned) 16 bytes with the double compare-and-exchange operation. OdinFS leverages this to update simple metadata whenever possible. For complex metadata updates, OdinFS leverages journaling for crash consistency. Note that OdinFS does not need to journal file data for its current consistency models. OdinFS inherits the per-CPU undo journal design from `WineFS` [25]. As detailed below, OdinFS ensures a file can only be in one per-CPU journal at any time. Hence, OdinFS can recover from the per-CPU journals by using a global transaction ID.

Ensuring crash consistency with range locks. Since the range lock allows multiple threads to access the same file (§3.2), ensuring crash consistency becomes a challenge. OdinFS addresses this issue by maintaining an invariant that a file can only be in one per-CPU journal. The key idea is that if a thread performs any operation that requires jour-

naling, it must acquire the writer lock of the whole inode lock for exclusiveness, even if this may reduce concurrency. Specifically, we classify data operations into three types: *read*, *overwrite* (writes to an existing file block), and *unallocated write* (writes to an unallocated file block, such as appending a file or writing to holes in a sparse file). Only the metadata stored in the inode, such as access or modification time, needs to be updated for read and overwrite. Following PMFS strategy, OdinFS stores these fields in a 16-byte PM block and updates them atomically without journaling. Hence, OdinFS can allow concurrent reads and overwrites to the same file. However, an unallocated write updates both inode and multiple blocks in the indexing structure and thus requires journaling. Hence, OdinFS only allows one thread to perform unallocated write at a time to maintain the invariance.

Thus, for reads, OdinFS acquires the reader lock of the whole inode lock and reader lock of the relevant range in the range lock. For writes, OdinFS distinguishes between overwrite and unallocated write. OdinFS first acquires the reader lock of the whole inode lock and walks the indexing structure to identify whether the write involves writing to unallocated blocks. No journaling is required if the write only updates allocated blocks (*i.e.*, overwrite). Hence, the thread can proceed by acquiring the writer lock of the relevant range in the range lock. Otherwise, it is an unallocated write and requires journaling. The thread then upgrades from the reader lock to the writer lock of the whole inode lock to ensure the inode is only in one per-CPU journal.

4 OdinFS implementation

File system implementation. We modify and extend PMFS [17] to design and implement OdinFS, while also referring to NOVA [41] and WineFS [25]. In summary, the inode table consists of multiple blocks forming a linked list. The directory data structure is similar to a linked list. The indexing structure of a regular file is a B-tree. Crash consistency is achieved with atomic updates and undo journaling (§3.7). OdinFS maintains the inode allocator, block allocator, and cached directory entries in DRAM with red-black trees. The state of the inode and block allocator needs to persist across power cycles. Thus, OdinFS writes their state to PM during unmount and reads from PM during mount. Upon crash, OdinFS recovers the state by scanning used inodes and their indexing structures. To minimize the synchronization overhead, inode allocator, block allocator, inode table, and journaling use per-CPU data structures. To handle complex metadata operations, such as `rename` or `mmap`, OdinFS follows PMFS by using the synchronization mechanisms in both VFS and the file system. We implemented OdinFS as a kernel module for the 5.13.13 Linux kernel and thus, its deployment challenges and manageability are similar to other in-kernel file systems. OdinFS is publicly available at https://github.com/rs3lab/Odinfs.

Efficient communication with delegation threads. Application and delegation threads communicate via a ring buffer (§3.3). To minimize communication overhead, we adopt the scalable ring buffer implementation from Solros [32]. Furthermore, each delegation thread has its private ring buffer to reduce the contention. The application threads send requests to a random delegation thread in the target NUMA node to load balancing delegation threads. We choose this algorithm since it incurs minimal runtime overhead without central coordination while achieving good performance. For PM access request notification, we use a pair of per-NUMA counters: issued counter and completed counter. The application thread increases the issued counter for each request, and sends the pointer of the completed counter. After issuing all the requests, the application thread waits until the number of issued request count on each NUMA node equals the per-NUMA completed count, which is atomically updated by delegation threads.

Accessing userspace memory via delegation. Delegation threads do not have access to the application address space, even though both of them are in the kernel space when handling a system call. We resolve this issue by first letting the application thread pre-faults and pins the user buffer pages in the kernel. It then passes the user buffer along with its root page table information (`mm->pgd`) to the delegation thread. Upon receiving the request, the delegation thread walks the page table for each user buffer page to figure out the physical page. Since the Linux kernel maps all physical pages into its address space linearly, the delegation threads can obtain the corresponding kernel virtual address by adding an offset. The delegationt threads can then access the user buffer with the kernel virtual address.

Minimizing synchronization overhead. To achieve the scalability design goal (§3.1), OdinFS further adopts state-of-the-art synchronization mechanisms to minimize the synchronization overhead. Specifically, OdinFS enhances the readers-writer range lock (§3.6) with BRAVO [15]. BRAVO optimizes the reader side performance of a readers-writer lock by leveraging a hash table, thus avoiding updating the shared reader counters. As discussed in §3.7, since a thread only acquires the writer lock of the whole inode lock for unallocated-write, we use the readers-writer semaphore in [30] for the whole inode lock. The per-CPU readers-writer semaphore optimizes the reader side performance of the semaphore with a per-CPU counter.

CPU usage fairness with delegation. To ensure fairness, OdinFS charges the request serving time of delegation threads to the application thread. Specifically, the application thread passes a pointer to its CPU usage time (`vruntime`) in each request, and the delegation threads thus atomically update it accordingly.

Implementation limitations. In the current implementation, we pin each delegation thread on a particular CPU.

Furthermore, with the current Linux scheduler, if application threads are also pinned to the same CPU, both the delegation and the application threads' performance will degrade. We plan to explore a lightweight and more efficient thread scheduling algorithm to address this issue.

5 Evaluation

We evaluate OdinFS by answering the following questions:

- How does opportunistic delegation affect OdinFS' performance? (§5.2)
- What is the I/O amplification factor of OdinFS? (§5.3)
- Does OdinFS scale with different delegation thread counts and PM NUMA nodes? (§5.4)
- Does OdinFS scale data operations? (§5.5)
- How does OdinFS perform with real-world applications? (§5.6)

5.1 Evaluation methodology

Evaluation environment. We conduct our evaluation on an eight-socket server. Each socket equips a 28-core Intel Xeon processor (224 cores in total) and six 128GB Intel Optane DIMMs interleaved at 4KB (768GB on each NUMA node). The machine has a total DRAM size of 768GB, with two A100 and two A5000 Nvidia GPUs. The server is running Linux kernel v5.13.13 and hyper-threading is disabled.

OdinFS configuration and target comparisons. Unless otherwise mentioned, we configure OdinFS to run on all eight NUMA nodes with twelve delegation threads on each NUMA node. We evaluate and compare OdinFS with four PM file systems: `ext4` [2], `PMFS` [17], `NOVA` [41], and `WineFS` [25]. We configure `ext4` with the DAX option and all the other file systems with the default setup. They provide weaker or the same level of consistency as OdinFS (§3.7). Since these four file systems operate on a single PM NUMA node, we further include one setup: `ext4(RAID0)`. Specifically, we create a RAID0 across all eight PM NUMA nodes using dm-stripe [1] and mount `ext4` on top of it. We cannot run RAID0 with the other file systems because unlike `ext4`, other PM file systems access the PM storage device by memory mapping it into the kernel address space and accessing it with load and store. The RAID0 device created by PM block devices does not support memory mapping to the kernel address space. Because of this, existing PM file systems crash at the time of mounting.

To the best of our knowledge, `ext4(RAID0)` is the only available setup that utilizes all the PM NUMA nodes.[3] However, we further emulate a non-existent setup: `NOVA(MN)` (`NOVA` with multiple nodes) to estimate the performance of a NUMA-aware `NOVA`. Specifically, we mount a single instance of the `NOVA` file system on each NUMA node and evenly distribute the testing files among instances.

Workload. Our workloads include a wide range of file system use cases, covering both data- and metadata-intensive ones. For microbenchmarks, we chose `fio` [4] and FxMark [31] to measure throughput, latency, and scalability, respectively. We configure `fio` to let each thread access a 1GB private file. We only show the results of sequential access due to space limitations and confirm that random access yields similar results. We evaluate `fio` with both small (4KB) and large (2MB) access sizes. For macrobenchmarks, we use Webserver, Fileserver, Videoserver, and Varmail in Filebench [3] and DNN checkpointing.

[3] `WineFS` crashed when mounting on multiple PM NUMA nodes on our server.

5.2 Throughput and latency

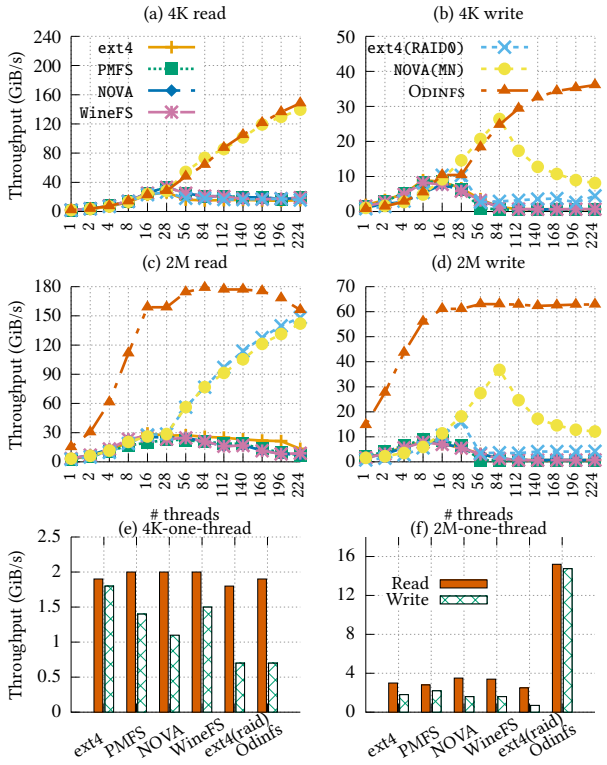


Figure 6: Throughput of evaluated file systems with up to 224 threads. OdinFS scales and outperforms others by up to 24.7×, as it utilizes all PM NUMA nodes and controls concurrent PM accesses.

Throughput. We use `fio` to evaluate OdinFS's throughput and latency. Figure 6 shows the throughput of all evaluated file systems. For `4K-read`, when the thread count is low ($\leq$ 28), all the evaluated file systems perform similarly. However, only OdinFS and `NOVA(MN)` scale beyond one NUMA node, outperforming other file systems by 9.4× with 224 threads. For `4K-write`, when the thread count is less than eight, OdinFS suffers from the communication overhead due to delegation and is up to 62% slower than other file systems. However, when the thread count reaches a certain threshold, the throughput of `ext4`, `PMFS`, and `NOVA` starts to collapse due to the reducing efficiency of XPBuffer and XPPrefetcher (§2.2). Instead, OdinFS can maintain its throughput thanks to limiting PM accesses, outperforming others by up to 8.1×.

With the 2MB access size, OdinFS benefits from accessing all PM NUMA nodes in parallel to serve IO requests. As a result, OdinFS outperforms other file systems even with a

low thread count. OdinFS allows applications to utilize most PM bandwidth with eight threads for write and 28 threads for read. OdinFS similarly scales both read and write throughput, outperforming other file systems by 1.1× to 24.7×, and up to 14.8× for `2M-read`, and `2M-write`, respectively. The throughput drop in OdinFS with `2M-read` is likely due to the increasing contention in the ring buffer or the shared counters §4. We plan to address this issue by investigating mechanisms to further increase the scalability of the communication mechanisms.

OdinFS scales because (1) OdinFS utilizes all the PM NUMA nodes in the system , and (2) it limits the number of PM access threads to avoid the performance collapse. `ext4(RAID0)` and `NOVA(MN)` similarly utilize all the PM NUMA nodes and thus performs closest to OdinFS. However, `ext4(RAID0)` and `NOVA(MN)` only scale read operations. `ext4(RAID0)` cannot scale `4K-read` due to a scalability bottleneck in small reads (§5.5). With `2M-read`, they only reach the throughput of OdinFS with a high thread count.

Summary: For small I/O requests, OdinFS incurs overhead with a small thread count but preserves PM performance with a large thread count. For large I/O requests, OdinFS benefits from handling them by paralleling accesses to PM NUMA nodes. This allows an application to utilize most of the PM bandwidth even if it has a small thread count. In summary, OdinFS scales both read and write operations for both small and large I/O sizes.

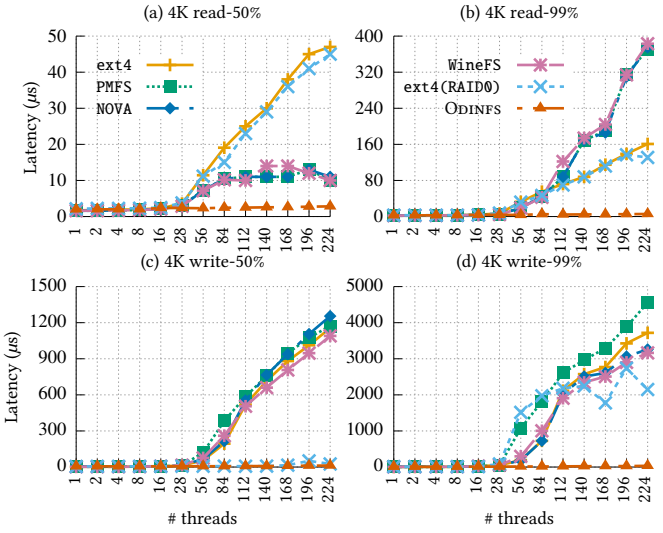


Figure 7: The median and 99 percentile latency of the evaluated file systems with a 4KB access size. OdinFS constantly maintains the low latency due to delegating PM accesses, avoiding the performance collapse within and across NUMA nodes.

Latency. Figure 7 presents the mean and the 99 percentile latency of all the evaluated file systems. For all the other file systems, increasing threads lead to either contention on PM or suffering from PM NUMA impact, resulting in skyrocketing latency. Thanks to delegation, OdinFS constantly maintains low latency. Its median and 99 percentile are 2.8µs and 5.7µs for `4K-read`, 5.4µs to 12.0µs and 6.3µs to 32.4µs for `4K-write`, respectively, outperforming the other file systems by up to 190×. OdinFS consistency has lower latency than `ext4(RAID0)`. The lowest median latency of other file systems is 1.6µs for `4K-read` and 2.6µs for `4K-write` with one thread. However, their latency quickly worsens after sixteen read threads and eight write threads. With a 2MB access size, the trend is similar. The performance advantage of OdinFS is even higher due to parallelization.

Summary : Delegating PM accesses enables OdinFS to maintain low latency.

5.3 IO amplification

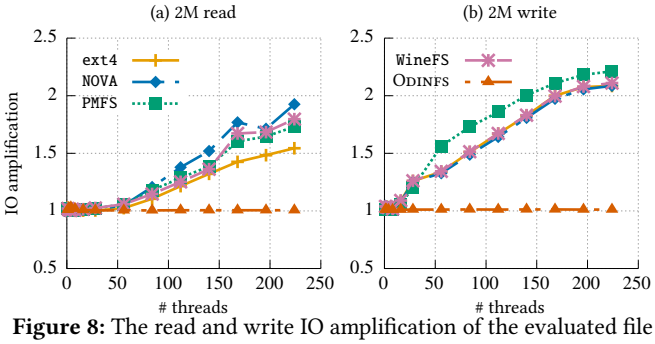


Figure 8: The read and write IO amplification of the evaluated file systems. OdinFS achieves low IO amplification since delegation maintains the caching/prefetching efficiency.

The conventional wisdom is that IO amplification is relevant for traditional storage devices (especially SSD) but not for PM since it is byte-addressable. However, due to the access size mismatch between the memory controller and the PM storage media, PM also suffers from IO amplification if the internal caching/prefetching becomes inefficient (§2.2). A high IO amplification reduces the PM lifetime and causes a latency spike triggered by the internal wear-leveling operation [43]. Thus, a PM file system must reduce it.

We report the I/O amplification as the number of bytes read from (or written to) the underlying PM media divided by the number of bytes requested (or issued) by the CPUs. We use Intel PMWatch [9] to obtain the relevant data. Figure 8 shows the I/O amplification for different file systems with the same setup in §5.2. OdinFS constantly achieves a low IO amplification (*i.e.*, less PM-level IO incurred for the same workloads) with increasing threads since delegation limits concurrent accesses and thus preserves the caching/prefetching efficiency. All other file systems suffer from a high IO amplification rate (*i.e.*, more PM-level IO incurred for the same workload), validating their low throughput (Figure 6) and high latency (Figure 7).

Summary : I/O amplification is still relevant for PM. OdinFS maintains a balance of amplification and high PM utilization. Our delegation scheme limits concurrent accesses, which maintains the caching/prefetching efficiency.

5.4 Sensitivity analysis

This section presents how delegation thread counts and PM NUMA nodes affect OdinFS's performance. We use the same experimental setup in §5.2.

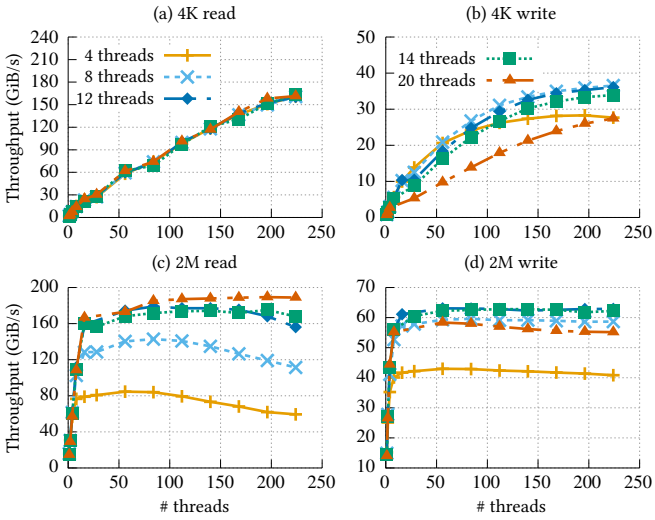


Figure 9: OdinFS's throughput with different delegation threads.

OdinFS with varying delegation threads. The optimal number of delegation threads for OdinFS depends on many factors, such as the relative speed between the processor and PM. Thus, we run experiments that vary the delegation thread counts to find the optimal one for our system.

Figure 9 shows the results. With `4K-read`, the delegation thread counts have no impact because OdinFS does not delegate read accesses (§3.5). With `2M-read`, the throughput is close to being saturated with twelve delegation threads but continues to increase until twenty delegation threads. With `4K-write` and `2M-write`, the throughput of OdinFS increases with up to eight or twelve delegation threads, respectively. Hence, we chose twelve delegation threads as the default setup for OdinFS since it performs well in all four setups.

Summary : The optimal delegation thread number in OdinFS depends on many factors and thus should be decided with experiments. Twelve delegation threads achieve a balanced performance in our system.

OdinFS with varying PM NUMA nodes. Figure 10 shows OdinFS's throughput with a different number of PM NUMA nodes. For `4K-read`, OdinFS enables delegation to prevent throughput collapse after 56 threads with one PM NUMA node and 112 threads with two PM NUMA nodes (§3.5). For the other three setups, OdinFS always delegates PM accesses. The results show that (1) OdinFS can maintain the throughput with a high thread count for different numbers of PM NUMA nodes, and (2) OdinFS scales PM performance with increasing PM NUMA nodes.

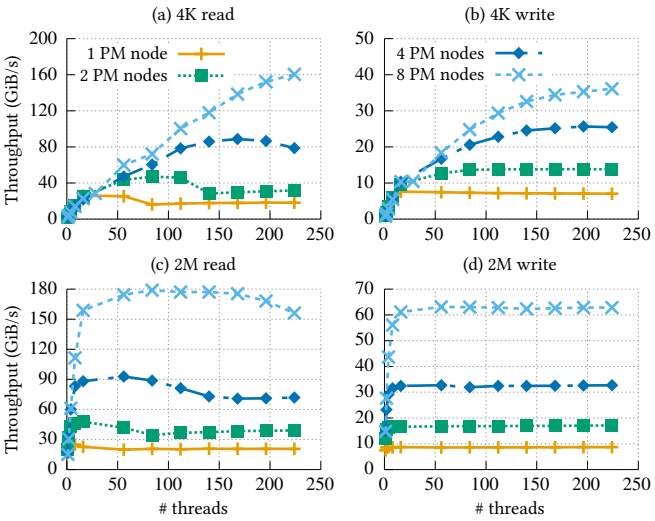


Figure 10: OdinFS with different numbers of PM NUMA nodes.

Name	Description
`DRBL`	Each thread reads a private block in a private file.
`DRBM`	Each thread reads a private block in a shared file.
`DRBH`	Each thread reads a shared block in a shared file.
`DWOL`	Each thread overwrites a private block in a private file
`DWSL`	Same as `DWOL` plus an `fsync()` after each writes
`DWAL`	Each thread appends to a private file
`DWOM`	Each thread writes to a private block in a shared file

Table 1: Summary of microbenchmarks in the FxMark suites [31]. Each thread repetitively performs the corresponding operations in each microbenchmark.

Summary : OdinFS scales PM performance with increasing PM NUMA nodes because of its efficient delegation scheme, showing the generality of its design.

5.5 Datapath scalability

To test whether OdinFS achieves the scalability design goal, we evaluate it with FxMark [31] microbenchmark suites. OdinFS mainly focuses on data operations and partially reuses the scalable data structures in `NOVA` and `WineFS` for metadata scalability. Hence, we focus on evaluating the scalability of data operations. Table 1 summarizes the FxMark microbenchmarks used in the evaluation. We use all the data operation microbenchmarks from FxMark except `DWTL`, where each thread concurrently truncates a private file; `DWTL` does not involve typical data operations (*i.e.*, read or write), and thus we view it as a metadata microbenchmark.

Figure 11 shows the scalability results of the evaluated file systems. Among the compared file systems, only `PMFS` and `NOVA` can scale one microbenchmark: `DRBL`. Instead, OdinFS scales all seven evaluated microbenchmarks. For read microbenchmarks, OdinFS is 12% slower than `PMFS` in `DRBL`. However, OdinFS outperforms other file systems by around 233× and 269× in `DRBM` and `DRBH`, respectively. For `DRBM` and `DRBH`, all other evaluated file systems suffer from the

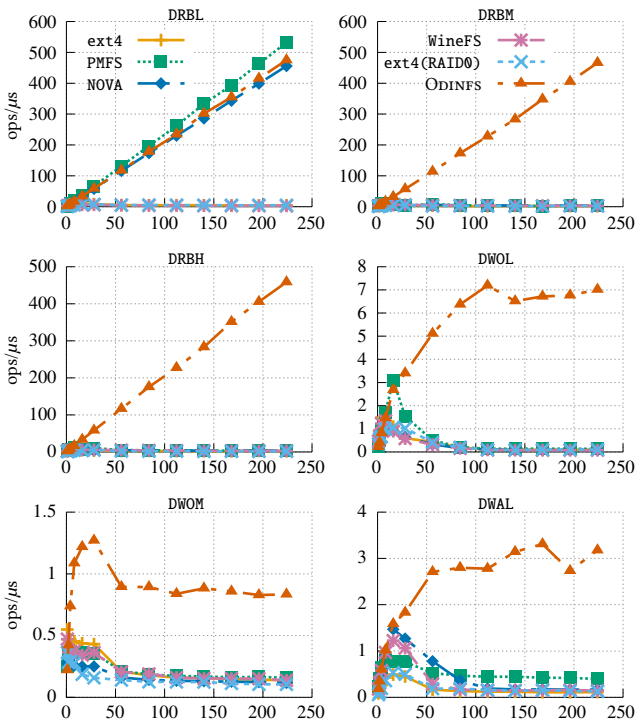


Figure 11: Scalability of data operations with the evaluated file systems. DWSL is not shown since its result is the same as DWOL. Other evaluated file systems only scale DRBL while OdinFS scales all microbenchmarks, thanks to controlling PM accesses, scalable metadata structures and concurrency control mechanisms, and the unique readers-writer range lock.

default readers-writer semaphore implementation in the Linux kernel, resulting in high locking overhead. The scalability of OdinFS comes from the unique scalable synchronization mechanisms (per-CPU readers-writer semaphore and BRAVO on top of the range lock) that minimize the synchronization overhead (§3.6, §4).

For write microbenchmarks, OdinFS outperforms other file systems by 53.8×, 8.2×, and 8.3× in DWOL (DWSL)[4], DWAL, and DWOM, respectively. OdinFS scales DWOL and DWSL due to arbitrating PM accesses and thus prevents performance collapse caused by concurrent writes. OdinFS scales DWOL due to arbitrating PM accesses and the scalable allocator design. In addition to arbitrating PM accesses and the scalable allocator, OdinFS scales DWOM with the readers-writer range lock.

Summary : While other evaluated file systems only scale DRBL, OdinFS scales all seven evaluated microbenchmarks with PM access control, scalable metadata structures, scalable concurrency control mechanisms, and the unique readers-writer range lock.

Name	# Files	Avg. file size	I/O size (r/w)	R/W
Fileserver	10K	2MB	1MB / 256KB	1:2
Webserver	20K	4MB	1MB / 256KB	10:1
Videoserver	226	512MB	1MB / 1MB	27:1
Varmail	100K	16KB	1MB / 16KB	1:1

Table 2: Configuration of the Filebench workloads. Fileserver, Webserver, and Videoserver is data-intensive with large I/Os. Varmail is metadata-intensive with small I/Os, representing the worst case for OdinFS. Webserver and Varmail are write-intensive while Fileserver and Videoserver are read-intensive.

5.6 Macrobenchmarks

We use a set of benchmarks from Filebench [3] as macrobenchmarks to evaluate OdinFS. We select four benchmarks: Fileserver, Webserver, Videoserver, and Varmail with configurations shown in Table 2. We configure Fileserver, Webserver, and Videoserver to work on relatively large files, reflecting the trend of growing sizes with these types of files. Varmail works on a large number of small files and performs small IO, representing the worst case for OdinFS. Webserver and Videoserver are read-intensive while Fileserver and Varmail are write-intensive. For Videoserver, since not all the threads are doing the same task, we measure the overall read and write throughput. For the other benchmarks, we measure the number of operations per second.

Figure 12 shows the result. For Fileserver, OdinFS outperforms other file systems by 4.8× to 25.3×. For Webserver, OdinFS outperforms all the single PM file systems and ext4(RAID0) by at least 3.8× and 1.6× to 3.1×, respectively. For Videoserver, OdinFS outperforms single PM file systems by around 6.6× and at least 5.4× for read and write throughput, respectively. OdinFS outperforms ext4(RAID0) by up to 2.3× for read throughput and around 7.3× for write. For these benchmarks, OdinFS's performance advantage comes from delegating PM accesses to preserve the maximum performance and utilizing the bandwidth of all the PM NUMA nodes. For read-intensive benchmarks: Webserver and Videoserver, ext4(RAID0)'s performance matches OdinFS with large thread counts, which is consistent with results in §5.2. However, OdinFS still outperforms ext4(RAID0) by 1.6× with 224 threads for Webserver. ext4(RAID0) achieves the same read throughput for Videoserver with high thread counts. However, this is because OdinFS still maintains around 4GiB/s write throughput while ext4(RAID0) completely starves the write threads.

Varmail is the worst case for OdinFS since delegating small I/Os incurs large communication overhead. However, the results show that OdinFS can maintain the similar performance as NOVA and WineFS. OdinFS outperforms PMFS, ext4, and ext4(RAID0) by 6.0× to 32.7×. The performance advantage of OdinFS, NOVA, and WineFS comes from the scalable

[4]The DWSL result is the same as DWOL since all evaluated file systems treat fsync() as a no-op.

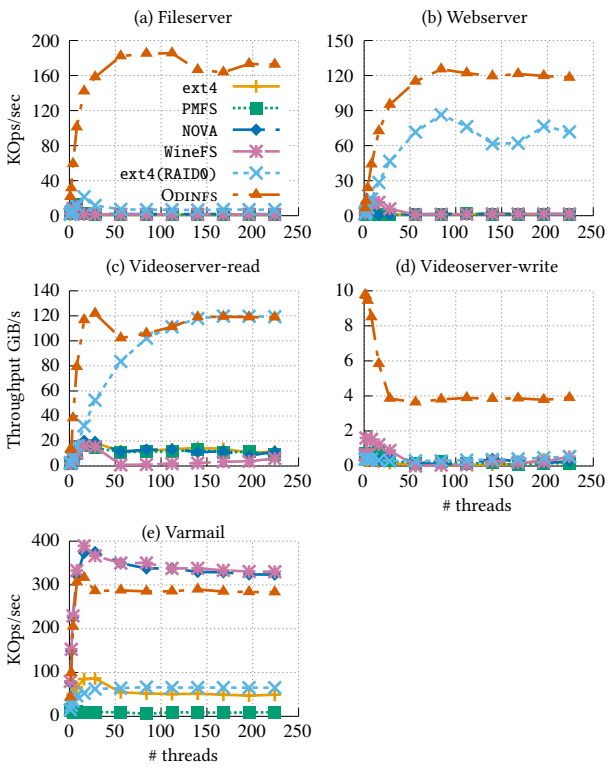


Figure 12: Results of the Filebench benchmarks. See Table 2 for configurations. Results show that OdinFS continue to scale PM performance with macrobenchmarks. OdinFS behaves the same as NOVA in the worst-case scenario: Varmail.

metadata structures (*e.g.*, in-memory directory indexing and per-CPU inode table).

Model	VGG16-ImageNet1K	BERT-SQuAD
Frequency (Steps/ckpt)	34	86
Checkpoint Size (MB)	1055	3828

Table 3: Machine learning checkpointing workloads setup. We use the same frequencies as [33]. A step refers to a training mini-batch.

Machine learning checkpointing. We also evaluate OdinFS with deep neural networks (DNN) checkpointing. DNN training is a time-consuming process and thus must checkpoint its state into persistent storage for fast failure recovery [33]. PM allows high frequency checkpointing and thus minimizes the window of losing work. Table 3 lists models and datasets we use. We measure the end-to-end execution time of training one epoch with checkpointing and the time spent in the file systems. Figure 13 shows the result. OdinFS results in end-to-end execution time reduction over the evaluated file systems by at least 2.6% on VGG16 and 12.3% on BERT. When looking into the time spent in the file systems, OdinFS outperforms evaluated file systems by at least 3.9× on VGG16 and 5.7× on BERT.

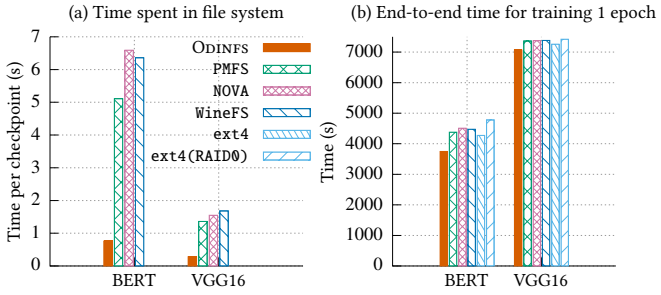


Figure 13: Machine learning checkpointing benchmark.

Summary : OdinFS continues to scale PM performance with macrobenchmarks. OdinFS achieves similar performance to state-of-the-art PM file systems in its worst-case scenario.

6 Discussion

6.1 PM I/O scheduling

The Videoserver result in §5.6 indicates that PM file systems similarly need to employ I/O scheduling as traditional file systems do to, for example, ensure fairness among applications [21, 44]. An I/O scheduling algorithm can be implemented in the system call, page cache, or block layer. Prior research has shown that an I/O scheduling algorithm is more effective if implemented across the various layers of the storage stack [44]. Existing PM file systems [2, 17, 25, 41] bypass the page cache and access PM directly with loads and stores. Hence, they can only implement the I/O scheduling algorithm in the system call layer, limiting the effectiveness of the scheduling algorithm.

Since OdinFS forces application threads to delegate bulk PM accesses to the delegation threads, the delegation threads thus become a central entity to access PM, acting similarly to the block layer in the traditional storage stack. Hence, the delegation enables OdinFS to perform I/O scheduling in both the system call layer and block layers. Exploiting this unique feature, we plan to extend OdinFS to support various I/O scheduling algorithms to ensure fairness or prevent head-of-line blocking to further improve its performance.

6.2 Comparison against RAID0

OdinFS's data layout policy is similar to RAID0 in that it stripes data across multiple PM NUMA nodes in a round-robin manner (§3.4). However, unlike RAID0, OdinFS employs PM-specific optimizations for small files (*i.e.*, put the first file stripe in the local PM of the creation threads). Furthermore, OdinFS stripes in the file level while RAID0 stripes in the disk level. OdinFS thus maximizes parallelization by ensuring adjacent file stripes are highly unlikely in the same PM NUMA node.

A fundamental difference between OdinFS and RAID0 is that OdinFS delegates PM access. As discussed across the paper, this enables OdinFS to avoid the bandwidth collapse with many access threads, minimize the PM NUMA impact,

and access PM in parallel, thus maximizing the PM performance. A typical RAID0 implementation achieves none of the above tasks. As a result, OdinFS consistently outperforms `ext4(RAID0)` with the evaluated benchmarks (§5).

6.3 Applicability to future PM hardware

Two of OdinFS's design goals: localized PM access and access arbitration, are based on the current implementation of PM hardware (§2). Although there is a possibility that future PM hardware may mitigate the above issues, we believe that many of OdinFS' designs will remain effective. Specifically, the PM NUMA impact is mostly due to the implementation of directory coherence information. Some of the recent Intel two-socket machines support snoop protocol for PM, and a prior work [27] has reported that the snoop protocol can significantly mitigate the PM NUMA impact. However, we find that many (large) multi-socket machines, only support directory coherence protocol. Hence, we believe that localized PM access is still a practically important design consideration.

Excessive concurrent access leads to performance collapse since it renders the on-DIMM caching and prefetching inefficient (§2.2). Furthermore, the access size mismatch between the CPU (64 bytes) and the underlying storage media (256 bytes) exacerbates the performance collapse. It also reduces the lifetime of the PM device due to the incurred I/O amplification. While it is difficult to predict the feasibility of changing the PM access sizes in future PM hardware, we expect that such changes would be non-trivial, especially making the PM access size as small as the CPU access size. This would require changes to other critical components in a PM DIMM, such as the address indirection table (AIT) and the DIMM-level prefetching logic. Furthermore, despite the reduced PM access size, limiting concurrent access might still be needed to avoid the performance degradation caused by DIMM-level cache thrashing.

In summary, we expect that both localized PM access and access arbitration will still be relevant for future PM hardware. Moreover, since OdinFS only incurs a small delegation overhead, it provides a "cost-effective" solution to the above problems without hardware changes. In addition, OdinFS' automatic parallelization design will remain useful to utilize the aggregated PM bandwidth across NUMA nodes without modifying the application code. We believe that the automatic parallelization design can be further generalized to other present or future storage systems (*e.g.*, CCIX-based storage systems [13]).

7 Limitations

Extra CPU usage. OdinFS's design incurs additional CPU usage due to parallelizing large PM accesses and the communication between the application thread and the delegation thread. OdinFS's current design reduces the CPU usage by pausing the delegation threads if there is no incoming request (§3.5). In addition, OdinFS can further trim down the CPU usage by (1) offloading PM accesses to I/OAT DMA and (2) disabling the delegation when there is no idle CPU on one NUMA node.

Stripping overhead. OdinFS stripes the data of a file across all NUMA nodes so that even a single-threaded application can benefit from the aggregated PM bandwidth through automatic parallelization (§3.2). However, since the stripping often involves remote access, while the delegation mechanism already significantly mitigates the NUMA impact, the stripping may reduce the best-case throughput and latency. Specifically, without stripping, a best-case scenario occurs when the application and the PM data are in the same NUMA node. However, benefiting from such a scenario requires extra code development to remember the NUMA node where the data resides and pin application threads to the NUMA node, which also limits scheduling flexibility.

If an application does not benefit from the automatic parallelization and wishes to enjoy the best-case performance, we expect OdinFS can work without stripping by placing the data of a file on a single NUMA node. In this case, OdinFS still achieves the other two design goals: (1) limiting concurrent accesses and (2) minimizing the PM NUMA impacts. Large I/O accesses can still be parallelized to one NUMA node but not all NUMA nodes as before.

Memory mapping. Due to stripping, OdinFS's memory mapping (mmap) performance is lower than other single-NUMA-node PM file systems in the best-case scenario as described above. To optimize this, OdinFS can use a copy-then-mmap model similar to `NOVA` [41]. Specifically, upon `mmap`, OdinFS allocates PM pages in the same NUMA node, copies the file content in remote NUMA nodes to these pages, and `mmap` the PM pages to the applications. Upon `msync` or `munmap`, OdinFS propagates the changes in the replicated PM pages back to the files.

8 Related work

PM file system. Unlike OdinFS, most existing PM file systems are designed to work on a single PM NUMA node and do not limit the number of access threads [12, 16, 17, 24, 28, 34, 41], leading to PM performance collapse. In terms of NUMA-aware PM file systems, Xu *et al.* proposed a new `ioctl` command that allows applications to specify the preferred NUMA node of a file [42]. This approach requires application changes and relies on the application to avoid the NUMA impact. `WineFS` [25] assigns a home NUMA node to each application thread and migrates the thread to the home NUMA node before writing to PM. As acknowledged by the authors, thread migration is expensive. Furthermore, it still suffers from the NUMA impact when two threads from different home nodes share the same file. Unlike OdinFS, none of these works focuses on utilizing both local and remote PM simultaneously as OdinFS does. `ext4(RAID0)` [1, 2] does not control the number of access threads nor resolve the PM NUMA impact, leading to lower performance than OdinFS in most cases.

Other NUMA-aware PM systems. PACTree uses the snoop protocol to minimize the PM NUMA impact [27]. However, the snoop protocol is unlikely to scale on large NUMA machines and may thus impact the performance of memory-intensive applications. Nap caches frequently accessed items in DRAM to avoid remote PM accesses [38]. However, Nap relies on a skewed access pattern to benefit from caching and still suffers from the PM NUMA impact upon cache misses. OdinFS proposes a fundamental different approach that uses delegation to address the PM NUMA impact.

Scalable file system. There are scalable file systems for both traditional storage devices [11, 30, 36] and PM [12, 41]. NOVA designs several scalable metadata structures, and OdinFS inherits them. Similar to OdinFS, there are file systems scaling the data operations with range locks [12, 36]. The closest work to OdinFS is KucoFS [12]. KucoFS is a PM file system that scales metadata operations through bypassing the VFS and scales data operations with range locks and versioned read. However, KucoFS shows that it cannot scale data operation benchmarks in FxMark beyond fifteen threads, while OdinFS scales all of them up to 224 threads. The difference is that (1) OdinFS uses delegation to prevent PM performance collapse while minimizing the NUMA impact. (2) OdinFS uses state-of-the-art concurrency mechanisms which minimize the synchronization overhead.

Localized I/O threads. Since PCIe devices also conform to NUMA topology, utilizing localized I/O threads (*i.e.*, placing I/O threads in the same NUMA code as the I/O devices) is a relatively common design in many non-PM systems [22, 35, 45]. To realize the old wisdom in PM systems, OdinFS has encountered and resolved many unexplored challenges (§3, §4), leading to a design significantly departs from the other PM systems (§3.2).

9 Conclusion

This paper presents OdinFS, a file system that maximizes PM performance in NUMA machines. A key novelty in OdinFS lies in decoupling the PM data accesses from the application threads by offloading them to a set of delegation threads in each NUMA node. Such decoupling simultaneously allows OdinFS to preserve the maximum PM performance with a single NUMA node, efficiently utilize PM in remote NUMA nodes, and service system calls by accessing PM in all NUMA nodes in parallel, thus maximizing the PM performance. OdinFS further includes fine-grained synchronization control mechanisms to scale all typical file system data operations. Extensive evaluation shows that OdinFS constantly outperforms existing PM file systems by several times to orders of magnitude.

Acknowledgments

We sincerely thank our shepherd Oana Balmau and the anonymous reviewers for their insightful feedback. Yunxin Sun contributed to the evaluation. This work was in part supported by Institute for Information and communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (No. 2014-3-00035).

References

[1] Device Mapper. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/striped.html.

[2] Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[3] Filebench - A Model Based File System Workload Generator . https://github.com/filebench/filebench.

[4] Flexible I/O Tester. https://github.com/axboe/fio.

[5] Directory Structure in Skylake Server CPUs, . https://community.intel.com/t5/Software-Tuning-Performance/Directory-Structure-in-Skylake-Server-CPUs/td-p/1185376.

[6] Intel® 64 and IA-32 Architectures Optimization Reference Manual, . https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[7] Intel Processor Counter Monitor (PCM), . https://github.com/opcm/pcm.

[8] Intel Optane Persistent Memory, . https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[9] Intel PMWatch. https://github.com/intel/intel-pmwatch.

[10] T. E. Anderson, M. Canini, J. Kim, D. Kostic, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, Nov. 2020.

[11] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[12] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, Feb. 2021.

[13] T. C. Consortium. Home Is Where the Memory Is, 2022. https://www.ccixconsortium.com/home-is-where-the-memory-is/.

[14] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 ACM SIGMOD/PODS Conference*, Xi'an, Shaanxi, China, May 2021.

[15] D. Dice and A. Kogan. BRAVO: Biased Locking for Reader-Writer Locks. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[16] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen. Performance and Protection in the ZoFS User-space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[17] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[18] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali. Single Machine Graph Analytics on Massive Datasets using Intel Optane DC Persistent Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, Aug. 2020.

[19] S. Gugnani, A. Kashyap, and X. Lu. Understanding the Idiosyncrasies of Real Persistent Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, Aug. 2020.

[20] T. Hirofuchi and R. Takano. The Preliminary Evaluation of a Hypervisor-based Virtualization Mechanism for Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1907.12014*, 2019.

[21] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021.

[22] Intel. Storage Performance Development Kit, 2021. SPDK.io.

[23] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714*, 2019.

[24] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[25] R. Kadekodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. R. Ganger, A. Kolli, and V. Chidambaram. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[26] S. Kashyap, C. Min, and T. Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[27] W.-H. Kim, R. M. Krishnan, X. F. S. Kashyap, and C. Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[28] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[29] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. Evaluating Persistent Memory Range Indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, Los Angeles, CA, Aug. 2019.

[30] X. Liao, Y. Lu, E. Xu, and J. Shu. Max: A Multicore-Accelerated File System for Flash Storage. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, Virtual, July 2021.

[31] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[32] C. Min, W.-H. Kang, M. Kumar, S. Kashyap, S. Maass, H. Jo, and T. Kim. SOLROS: A Data-Centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, Porto, Portugal, Apr. 2018.

[33] J. Mohan, A. Phanishayee, and V. Chidambaram. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, Feb. 2021.

[34] I. Neal, G. Zuo, E. Shiple, T. A. Khan, Y. Kwon, S. Peter, and B. Kasikci. Rethinking File Mapping for Persistent Memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, Feb. 2021.

[35] T. L. F. Projects. DPDK, 2021. https://www.dpdk.org/.

[36] Y. Ren, C. Min, and S. Kannan. CrossFS: A Cross-layered Direct-Access File System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, Nov. 2020.

[37] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent Memory I/O Primitives. In *Proceedings of the International Workshop on Data Management on New Hardware*, Amsterdam, The Netherlands, July 2019.

[38] Q. Wang, Y. Lu, J. Li, and J. Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021.

[39] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao. Characterizing and Modeling Non-Volatile Memory Systems. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Virtual, Oct. 2020.

[40] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons. An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Denver, CO, Nov. 2019.

[41] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, Feb. 2016.

[42] J. Xu, J. Kim, A. Memaripour, and S. Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the 23th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.

[43] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, Feb. 2020.

[44] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[45] D. Zheng, R. Burns, and A. S. Szalay. Toward Millions of File System IOPS on Low-Cost, Commodity Hardware. In *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Denver, CO, Nov. 2013.

DURINN: Adversarial Memory and Thread Interleaving for Detecting Durable Linearizability Bugs

Xinwei Fu
Virginia Tech

Dongyoon Lee
Stony Brook University

Changwoo Min
Virginia Tech

Abstract

Non-volatile memory (NVM) has promoted the development of *concurrent crash-consistent* data structures, which serve as the backbone of various in-memory persistent applications. Durable linearizability defines the correct semantics of NVM-backed concurrent crash-consistent data structures, in which linearizability is preserved even in the presence of a crash event. However, designing and implementing a correct durable linearizable data structure remain challenging as developers are to manually control durability (persistence) using low-level cache flush and store fence instructions.

We present DURINN, to the best of our knowledge, the first durable linearizability checker for concurrent NVM data structures. DURINN is based on the novel observation on the gap between linearizability point – when the changes to a concurrent data structure become publicly visible – and durability point – when the changes become persistent. From the detailed gap analysis, we derive three durable linearizability bug patterns that render a linearizable data structure not durable linearizable. To tame the huge NVM states and thread interleaving test space, DURINN statically identifies likely-linearization points and actively constructs adversarial NVM state and thread interleaving settings that increase the likelihood of revealing durable linearizability bugs. DURINN effectively detected 27 (15 new) durable linearizability bugs from 12 concurrent NVM data structures without a test space explosion problem.

1 Introduction

Non-volatile memory (NVM) is becoming widely adopted in various computer systems thanks to its storage-and-memory-like characteristics. Like storage, NVM is persistent across a power cycle and has a high density. Like memory, NVM provides byte-addressability and low-latency properties. A program can persist data in NVM using `load` and `store` instructions without paying storage stack overhead. Notably, Intel's Optane DC Persistent Memory [13, 47] has already been deployed in cloud [3] and supercomputer [2]. ARM also has announced its support for NVM [12, 14]. The upcoming Compute Express Link (CXL) [20] standard introduces cache-coherent CXL-attached NVM card with on-device cache.

The persistence and low-latency properties of NVM have promoted the development of various NVM-backed *concurrent crash-consistent* data structures, which serve as the key enabler of the application-level crash-consistency guarantee. For instance, concurrent NVM hash table is the backbone of NVM-backed `memcached` [7] and `redis` [10]. Concurrent NVM B+tree and hash map are the cores of `pmemkv` [4]. Concurrent NVM B-trees are used in NVM-backed file systems [19, 24–26]. In the event of an application or system crash, or a sudden power failure (a crash hereafter for brevity), an NVM program built on crash-consistent data structures can seamlessly resume its execution from the (recovered) NVM state as if nothing has happened.

Durable linearizability [40] defines the correct semantics of NVM-backed concurrent crash-consistent data structures, as linearizability [33] is the norm correctness standard for traditional (non-NVM) concurrent data structures. At a high level, durable linearizability requires that the effects of completed operations before a crash should remain completed and visible (like linearizability). Additionally, durable linearizability requires that the operation upon a crash be either fully executed ("all" semantic) or not at all executed ("nothing" semantic). However, designing and implementing correct durable linearizable data structures remain very challenging.

The fundamental challenge in developing a durable linearizable NVM data structure lies in *the gap between the linearization point (visibility) and the durability point (persistence).* Concurrent data structures use a synchronization operation (*e.g.*, `compare-and-swap` (`CAS`) in lock-free ones and `lock`/`unlock` in lock-based ones) as *the linearization point* to make one thread's effect visible to other threads. However, the completion of a synchronization operation does not guarantee durability. When the new value of a store (or `CAS`) instruction reaches a cache, it becomes visible, but it is not yet durable until it is written back to the NVM. A data staying in a volatile

cache[1] (as a dirty line) is lost upon a crash, and a cache may evict cache lines in an arbitrary order that is different from the program (store) order. As a result, *the durability point* may come later in time and may appear in a different order with respect to the preceding stores within the same thread or the remote loads from other threads. To ensure durable linearizability, developers have to manually control durability using cache line flush and store fence instructions (*e.g.*, `clwb` and `sfence` in x86 architecture), making durable linearizable NVM programming error-prone.

Unfortunately, existing solutions are not sufficient in testing durable linearizability of concurrent NVM data structures. Prior linearizability testing tools [15, 63] do not consider crash and recovery semantics. NVM-specific crash consistency testing tools either require user-defined custom oracles [22,31,35,43,45,46,51] or are limited to single-threaded NVM programs [30]. It is non-trivial to extend them for durable linearizability because testing space grows exponentially in two dimensions: *crash states* and *thread interleaving*.

This paper presents DURINN, an active and scalable durable linearizability checker for concurrent NVM data structures. DURINN is based on the novel observation on the gap between linearizability point where the changes to a data structure becomes visible and durability point where the changes become persistent and thus remain visible even after a crash. After analyzing what could go wrong if a crash occurs before, after, or between the linearizability and durability points, we derive three durable linearizability (DL) bug patterns that render a linearizable data structure not durable linearizable.

To tame the huge test space, DURINN uses two novel techniques: 1) *adversarial crash state and thread interleaving construction*, and 2) *likely-linearization point inference*. DURINN serves as an adversary of the three DL bug patterns, and actively constructs adversarial crash scenarios that specify which stores to (or not to) persist and which thread interleaving to consider. The intuition behind adversarial crash state construction is to maximize the difference between a constructed crash state and a consistent state preserving DL conditions, thus increasing the likelihood of revealing DL bugs. Furthermore, DURINN employs static program analysis to identify *likely-linearization points* and focuses on testing a program crash before and after those linearization points.

We evaluate DURINN with 13 concurrent NVM data structures, which are highly optimized for NVM and have shown to be more scalable than (simple) NVM hash tables and B-trees used in `memcached`, `redis`, `pmemkv`, etc. DURINN detected 27 (15 new) durable linearizability bugs in 12 data structures. 7 of 15 new bugs have been confirmed by the developers so far. Our evaluation also shows that DURINN can detect concurrent DL bugs (better detection effectiveness) with fewer tests (better scalability), compared to Witcher [30], the state-of-the-art NVM crash-consistency bug detector.

[1] We discuss the implications of (future) persistent cache later in §7.2.

The paper makes the following scientific contributions:

- We present three durable linearizability bug patterns after performing detailed analysis on how a linearizable data structure may violate durable linearizability.
- To our best knowledge, DURINN is the first durable linearizability checker designed for concurrent NVM data structures. The proposed adversarial crash state and thread interleaving construction and likely-linearization point inference allow DURINN to detect DL bugs in an active and scalable manner.
- DURINN reports 27 (15 new) bugs and outperforms the state-of-the-art NVM testing tool in terms of bug detection effectiveness and test space reduction.

2 Background

In this section, we first provide background on linearizability (§2.1) and durable linearizability (§2.2), and then discuss the persistence model used in this paper (§2.3).

2.1 Linearizability

Linearizability [33] is the widely-used correctness standard for concurrent data structures. Formally, linearizability is defined over an existence of an equivalent legal sequential history. Informally, a concurrent data structure is *linearizable* if each operation appears to take effect instantaneously at some moment between the operation begin and end events. If one operation precedes another, then the earlier operation must have taken effect before the next one. If two operations overlap, then their order can be serialized in any arbitrary order. Some pending operations can be thought to be complete.

A *linearization point* (LP) is a program point where an operation takes effect and its effects become visible to other operations. In a lock-based data structure, a critical section (or an `unlock` point) often serves as the linearization point. In a lock-free data structure, the linearization point is typically a single-step atomic instruction (*e.g.*, `CAS`) that makes its change visible to others. We refer to the variable used to make an operation's effect visible as a *synchronization variable* (SV). Atomically updating SV is a linearization point for a writer operation (*e.g.*, `insert`), while reading SV is a linearization point for a reader operation (*e.g.*, `get`).

2.2 Durable Linearizability

Durable linearizability [40] extends linearizability with the notion of a crash. With durable linearizability, a history may include a system-wide crash event (which does not belong to a specific thread) in addition to the operation begin and end events. The definition of precedence order is also extended to incorporate a crash. In durable linearizability, an operation makes its effects visible to others at the linearization point (like linearizability). Additionally, an operation makes its effects persisted at the *durability point* (DP) so that its effects remain completed and visible after a crash.

A *completed operation* refers to an operation whose instructions are all executed, end event is observed, and result

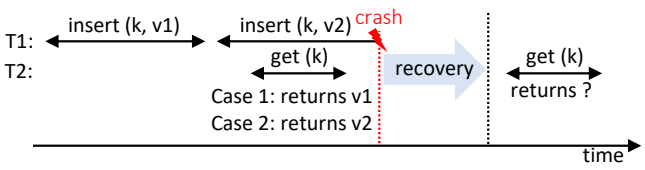


Figure 1: Durable linearizability example. The result of the second `get` (after a crash) depends on the result of the first `get` (before a crash), which also determines if the crashed `insert` takes effect.

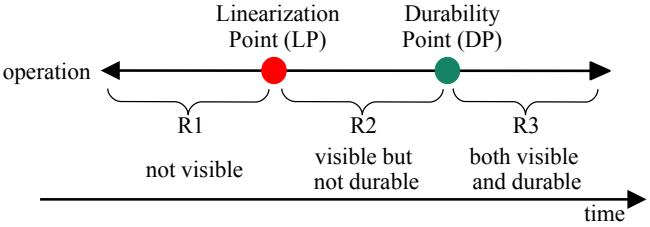


Figure 2: Linearization point and durability point split an operation into three-time intervals as per its visibility and durability guarantees.

is returned. Some crash-consistent programs that can recover from inconsistent NVM state using a custom crash consistency protocol may not require a completed operation to be fully durable.

Durable linearizability requires the following conditions:

- **(C1)** Without a crash, all operations are linearizable.
- **(C2)** If a crash happens, all previously completed operations (before a crash) should remain completed and their effects should remain visible after a crash.
- **(C3)** The operations that have not completed upon a crash could be considered either completed or not. When considered completed, its effect should be visible. In other words, the crashed operation should provide all-or-nothing semantics (fully executed or not at all executed).

Figure 1 illustrates a durable linearizable example. Thread `T1`'s first `insert` completes before a crash, so it should remain visible even after a crash by (C2). If `T2`'s first `get` returns `v1` before a crash, the second `get` after the crash could return either `v1` or `v2`, with a flexibility to follow all or nothing semantics in (C3). However, if `T2`'s first `get` returns `v2` (*i.e.*, it completes) before a crash, then the second `get` after the crash should return `v2`, according to (C1) and (C2).

Prior works such as Line-up [15] and Round-up [63] have demonstrated the practicality of (C1) linearizability testing. DURINN assumes a data structure under test is linearizable without a crash, and focuses on checking if it satisfies the (C2) and (C3) conditions in the presence of a crash.

2.3 Persistence Model

This paper assumes a volatile cache. This is the case for the current Intel x86 architecture with Optane NVM [39, 58] and ARM [12, 14]. The future Compute Express Link (CXL) [20] standard also introduces a cache-coherent interconnect and a CXL-attached NVM card with a volatile on-device cache. We will discuss the implications of persistent cache later in §7.2.

Given a volatile cache, dirty cache lines are lost upon a crash. To control durability, Intel provides cache flush instructions such as `clflush`, `clflushopt`, and `clwb`. When the asynchronous flush `clwb` instruction is used for performance, the store fence `sfence` instruction should be used together to ensure the completion of preceding `clwb` instructions [58]. Similarly, ARM supports `dc cvap` cache flush and `dsb` fence instructions [12, 14]. CXL introduces Global Persistent Flush (GPF) to enforce the persistence ordering on emerging CXL-attached NVM card [20].

3 Durable Linearizability Bugs

This section discusses the gap between linearization point and durability point (§3.1), and presents three durable linearizability bug patterns derived from the gap analysis, along with real-world examples detected by DURINN (§3.2-§3.4).

3.1 The Gap Between LP and DP

As illustrated in Figure 2, the duration of an operation can be partitioned into three regions (`R1`, `R2`, and `R3`) based on the linearization point (LP) and the durability point (DP). At LP, the effect of an operation becomes visible to other threads. At DP, the effect of an operation becomes durable (persisted) so that it can survive a crash and remain visible after a crash.

The bug patterns presented in this section assume that LP is known given an operation. We later in §5.2 discuss the static methods we used to identify *likely-linearization points*. For example, a lock-free `insert()` operation often uses an atomic instruction on a synchronization variable to make its effect visible in a single step. The atomic update (*e.g.*, `CAS`) forms LP, and the following cache line flush and fence instructions (*e.g.*, `clwb` and `sfence`) become DP.

The gap between LP and DP leads to different visibility and durability guarantees. Before LP (region `R1`), the effect of an operation is neither visible nor durable. Between LP and DP (region `R2`), the effect of an operation is visible but not durable. After DP (region `R3`), the effect of an operation is visible as well as durable. Durable linearizability defines different correct/wrong behaviors depending on when a crash occurs: after DP (region `R3`), before DP (regions `R1` and `R2`), and between LP and DP (region `R1`). From the classification, we derive the following three DL bug patterns.

3.2 DL Bug Pattern 1: An Incompletely-Durable Bug

The first *Incompletely-Durable* bug pattern considers a crash *after DP* (in region `R3`). As a crash happens after DP, all the changes made by the crashed operation should be completed and persisted as if no crash has happened. In other words, the crashed operation should provide the "all" (fully-executed) semantic guarantee. After resuming from a crash, if another operation may observe *incompletely durable* effects, then it may produce wrong output violating durable linearizability. Figure 3 illustrates the Incompletely-Durable bug pattern. Since the crash happens after DP of `T1`'s `insert(K,V)`, to be

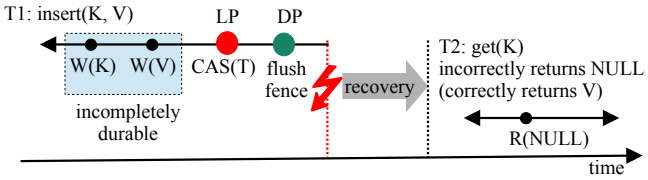


Figure 3: An *Incompletely-Durable* bug pattern. If a crash occurs after DP, the `insert` operation should make all its effects durable completely. Otherwise, the `get` operation after a crash may not be able to see its effect, producing wrong output.

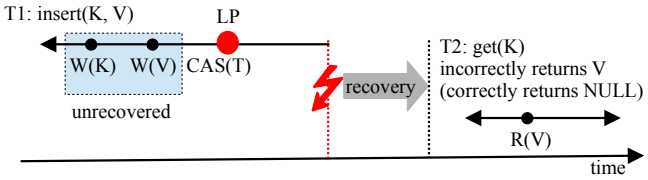


Figure 4: An *Unrecovered-Durable* bug pattern example. If a crash happens before DP, the `insert` operation should recover (undo) any partially durable effect. Otherwise, the `get` operation after a crash may see its partial effect, producing wrong output.

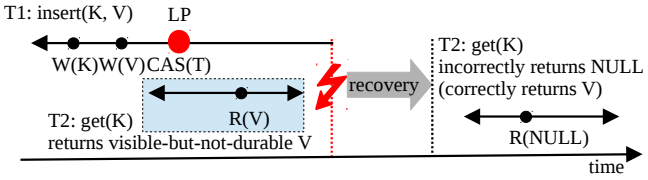
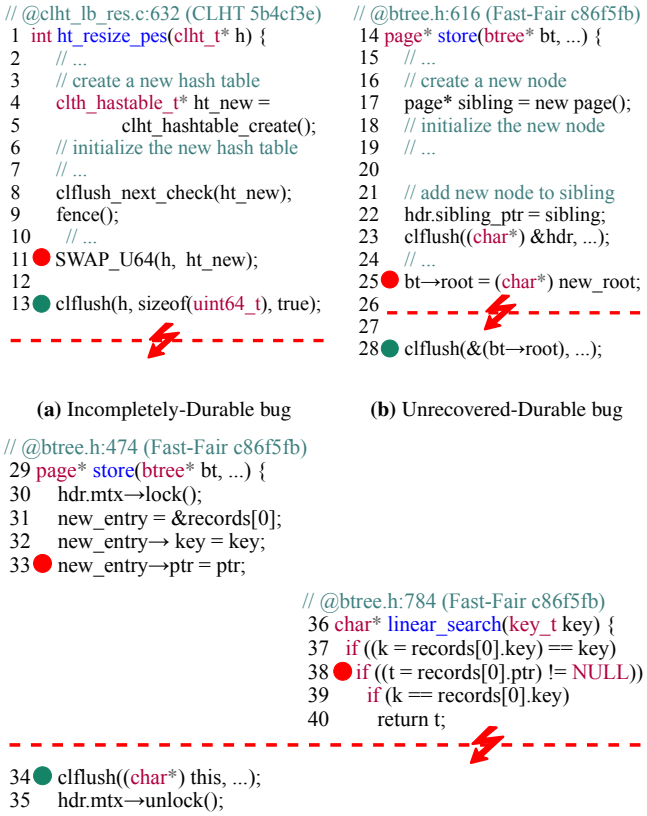


Figure 5: A *Visible-But-Not-Durable* bug pattern. For a crash between LP and DP of `insert`, the concurrent `get` may observe and return the visible-but-not-durable value. However, the second `get` after a crash may not be able to return the same value as the effect of `insert` is not durable.



(a) Incompletely-Durable bug

(b) Unrecovered-Durable bug

(c) Visible-But-Not-Durable bug

Figure 6: Durable linearizability bug examples in P-CLHT [44] (a) and Fast-Fair [34] (b) and (c). A red circle represents LP; green represents DP; and a red lightning bolt represents a crash.

durable linearizable, `T2`'s `get(K)` should return `V` after the recovery.

To avoid *Incompletely-Durable* bugs, all the preceding stores must be persisted before the store (or `CAS`) on a synchronization variable becomes persisted (DP), using cache line flush and fence instructions. This is analogous to the linearizability programming idiom in which all the preceding stores must be visible before the store (or `CAS`) on a synchronization variable become visible (LP), using a fence instruction.

Figure 6(a) shows a part of rehashing code in P-CLHT [44], a concurrent NVM hash table. The code first allocates a new hash table (line 4), updates/persists the new hash table (lines 6-7), and then atomically sets the root pointer `h` to the new hash table, making its effect visible (line 11, which is LP). However, `clflush_next_check` at line 8 does not flush all the updated NVM data in the new hash table and leaves a part unpersisted (an Incompletely-Durable bug). If a crash happens after DP – after persisting the root pointer `h` (line 13), the inserted key-value data after a crash may be lost, violating durable linearizability.

3.3 DL Bug Pattern 2: An Unrecovered-Durable Bug

The second *Unrecovered-Durable* bug pattern considers a crash *before DP* (in regions `R1` and `R2`). As a crash happens before DP, any temporal change made by the crashed operation should not be visible after the resumption. That is, the crashed operation should support the "nothing" (not-at-all-executed) semantic. After resuming from a crash, if another operation may observe *unrecovered durable* effects, it may produce wrong output violating durable linearizability. Figure 4 illustrates the Unrecovered-Durable bug pattern. Since the crash happens before DP of `T1`'s `insert(K,V)`, to be durable linearizable, `T2`'s `get(K)` should not return `V` after the recovery.

To avoid *Unrecovered-Durable* bugs, a durable linearizable data structure may opt to buffer/undo the effects of preceding stores before DP, or embed a custom logic to safely ignore partial NVM updates: *e.g.*, read key `K` and value `V` only if token `T` is set. This pattern is called "guarded protection" [30] and we discuss it in detail in §5.2.

Figure 6(b) shows an *Unrecovered-Durable* bug from Fast-Fair [34], a lock-based NVM B+tree. While splitting a node, it first creates a new node (line 17) and initializes the new node

(lines 18-19). Then it adds the new node to the sibling of the current node (line 22) and persists the change (line 23). Later, it sets the new root (line 25, which is LP). The `new_root` in line 25 is the new root node of the B+tree after a node split. The node, which the `hdr` belongs to, is a child of the new root. If a crash happens before persisting the new root node (line 28, DP), the B+tree will be in an illegal state in which the root node has a sibling node. Any further operation leads to a program crash and will lose all previously completed operations, violating durable linearizability.

3.4 DL Bug Pattern 3: A Visible-But-Not-Durable Bug

The last *Visible-But-Not-Durable* bug pattern considers a crash *between LP and DP* (in region `R2`). If a crash happens between them, the effect of the current operation may be *visible but not yet durable*. As it is visible, another concurrent operation can see the effect and take an action based on the observation: *e.g.*, returning a non-durable value.

Figure 5 illustrates an example. While thread `T1` is performing an `insert(K,V)` operation and just finishes executing its LP but not DP, thread `T2` performs a concurrent `get(K)` operation. The concurrent `get(K)` sees the non-durable effect of `insert(K,V)` and returns the value `V`. As the `get(K)` is completed before a crash, to be durable linearizable, `T2`'s second `get(K)` after the recovery should return `V` as well, but it cannot as the effect of `insert(K,V)` has not been persisted. Note that Visible-But-Not-Durable bugs may also occur between concurrent writers, say two `insert(A)` and `insert(B)` operations in a sorted linked list. The later `insert(B)` operation in the linearizable order may see the effect of the earlier `insert(A)` operation, adding `B` after `A`. Durable linearizability may be violated if a crash occurs after `insert(B)` completes but before `insert(A)` finishes.

To avoid *Visible-But-Not-Durable* bugs, an operation (later in the linearizable order) may be designed to wait until the earlier operation passes its DP. Alternatively, a lock-free design may use a "persistence-helping" mechanism [21, 28]. Suppose operation `A` updated `X` but did not persist it yet. Another concurrent operation `B` wants to take actions (*e.g.*, takes a different branch, persists other data) based on the value of `X`. Then `B` "helps" persist `X` on behalf of `A`. If a lock-free data structure does not implement a similar helping mechanism correctly and if `B` relies on unpersisted updates from `A`, then a Visible-But-Not-Durable bug may happen. The helping logic is analogous to the linearizability programming idiom in which one thread helps fix temporal inconsistency on behalf of another thread.

Figure 6(c) shows a *Visible-But-Not-Durable* bug from Fast-Fair [34]. The left code (`store`) and the right code (`linear_search`) are parts of `insert` and `get` operations, respectively. An `insert` operation first acquires the lock (line 30) then writes `key` and `ptr` (lines 31-33). It then persists the writes (line 34) and releases the lock at the end (line 35). Since Fast-Fair allows concurrent (non-blocking) `get` operations while splitting a node, linking a new node is LP for `insert` (line 33). On the other hand, the `get` operation refers to `ptr` (line 38, which is LP for `get`) while checking if there is any key change in-between by reading it twice (lines 37, 39). Suppose `linear_search` is scheduled between the LP (line 33) and DP (line 34) of `store` as shown in the figure. The concurrent `get` operation can read visible-but-not-durable data. If the crash happens before `insert`'s DP (line 34). After the recovery, the previously returned data cannot be accessed anymore because unpersisted data will be lost upon a crash. Thus, Fast-Fair violates durable linearizability.

4 Overview of Our Approach

In this section, we will first discuss the huge testing space as the main challenge in detecting durable linearizability bugs (§4.1). We then provide an overview of our two major techniques – (1) adversarial NVM state and thread interleaving construction (§4.2) and (2) likely-linearization point inference (§4.3) – designed to address the test space challenge.

4.1 Challenges in Detecting DL Bugs

Existing solutions are not sufficient in testing durable linearizability of concurrent NVM data structures. Traditional linearizability testing tools, such as Line-up [15] and Round-up [63], do not consider crash and recovery semantics. Most NVM-specific crash consistency bug detection tools (*e.g.*, Yat [43], PMTest [46], XFDetector [45], Agamotto [51], Jaaru [31], and PMDebugger [22]) are not designed for durable linearizability, and instead require user-defined custom oracles or consistency checkers. Some (*e.g.*, Witcher [30]) are limited to testing single-threaded NVM programs. We discuss related work in detail in §9.

It is non-trivial to extend existing NVM testing tools such as Yat and Witcher for durable linearizability because testing space grows exponentially in two dimensions: *NVM crash states* and *thread interleaving*. The crash state test space is huge since a crash can happen any time during an execution and a volatile cache can evict cache lines in an arbitrary order. For example, Yat [43], an exhaustive crash consistency testing tool, attempts to test 10^{31} crash states for an NVM hash table with 2000 operations [30]. Moreover, the number of thread interleaving grows exponentially (n^k) with the number of threads (n) and the number of steps (k) in each thread.

4.2 Adversarial NVM State and Thread Interleaving

We propose an adversarial technique to effectively explore the huge testing space in finding durable linearizability bugs. Instead of exhaustively or randomly exploring the testing space, we actively construct adversarial NVM states and adversarial thread interleavings, which are likely to trigger the three DL bug patterns discussed in §3. To the best of our knowledge, DURINN is the first work using an adversarial testing approach for bug detection in NVM programs.

Adversarial NVM state construction. For each DL bug pattern and a given crash location (*e.g.*, before or after DP),

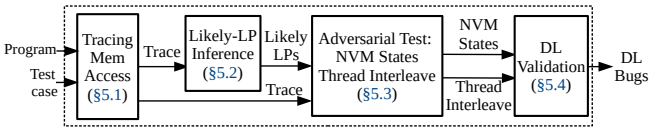


Figure 7: The overall architecture of DURINN.

DURINN determines which preceding stores should be or should not be persisted to increase the likelihood of triggering the DL bugs. For example, when testing an Incompletely-Durable bug after DP, DURINN adversarially constructs the (worst-case) NVM state where an update on a synchronization variable is persisted (at DP) but the preceding stores are not as persisted as possible. This way, DURINN can maximize the incompleteness of durability of a target operation, increasing the chance to break its "all" (fully-executed) semantic guarantee. Note that DURINN constructs only feasible NVM states while obeying the persistence model of a processor and program order semantics (*e.g.*, TSO for x86).

Adversarial thread interleaving construction. DURINN constructs adversarial thread interleaving only when thread interleaving is indispensable to trigger the DL bug patterns. The Incompletely-Durable and Unrecovered-Durable bugs do not require concurrent operations to trigger, so DURINN tests those bug patterns in a single-threaded mode. On the other hand, Visible-But-Not-Durable requires concurrent operations. The main challenge in triggering the Visible-But-Not-Durable bug is that two (or more) concurrent operations must be precisely scheduled in a very narrow window between LP and DP. DURINN adversarially constructs a thread interleaving such that a concurrent operation is scheduled between LP and DP of another operation.

4.3 Likely-Linearization Point Inference

Our adversarial NVM state and thread interleaving construction requires the knowledge of LP locations. Manual annotation of LPs would be error-prone and it makes DURINN not automatic. A naive approach, considering all stores as LPs, would lead to too many tests.

To address the problem, DURINN infers likely-LPs from source code based on the common concurrent NVM programming practices: (1) atomic instructions are used in concurrent programs for synchronization; (2) concurrent programs usually make a memory region visible to other threads after initialing the memory region; (3) NVM programs usually use guarded-protection [30] to ensure persistence atomicity. DURINN employs static program analysis to identify the above programming practices and infer likely-LPs. They are then fed to our adversarial NVM state and thread interleaving construction. The inferred likely-LPs are not necessary to be precise. A false positive LP will only lead to more tests. As far as we know, DURINN is the first work that statically infers linearization points from concurrent NVM programs.

```
// writer
 1  *key_ptr = key;
 2  *val_ptr  = val;
 3  flush(key_ptr);
 4  flush(val_ptr);
 5  fence();
 6
 7  flag = 1; // set guardian
 8  flush(&flag);
 9  fence();

// reader
10  // guardian read
11  if ( flag == 1)
12     func(key_ptr, val_ptr);
```

(a) Guarded-Protection

```
 1  // Memory allocation
 2  Node* new_node = alloc(sz);
 3  // Initialization
 4  new_node→key = key;
 5  new_node→val =  val;
 6  new_node→next = NULL;
 7  flush(&new_node→key);
 8  flush(&new_node→val);
 9  flush(&new_node→next);
10  fence();
11
12  // Add node to the core
13  core→tail = new_node;
14  flush(&core→tail);
15  fence();
```

(b) Publish-after-Initialization

Figure 8: Examples for *Guarded-Protection* and *Publish-after-Initialization* from (a) CCEH [50] and (b) NVTraverse [28]. Likely-linearization points are at line 7 and 13 in (a) and (b), respectively.

5 Design of DURINN

We present the overall architecture of DURINN at Figure 7. DURINN takes as input a target NVM data structure and a test case (a sequence of operations, such as `insert`, `delete`, and `get`) and reports detected durable linearizability bugs. DURINN first instruments a program and runs a test case to collect a memory trace (§5.1). DURINN then infers likely-linearization points from the trace (§5.2). Given the memory trace and the identified likely-linearization points, DURINN performs adversarial NVM state and thread interleaving construction (§5.3) to generate a collection of crashed NVM images and thread schedules to test. Lastly, DURINN validates the generated crashed NVM images along with the generated thread schedules to detect durable linearizability bugs (§5.4).

5.1 Tracing Memory Accesses

DURINN instruments all NVM memory accesses (load, store[2]) and NVM heap allocation. DURINN also traces control flow transfers (branch, function call) because our likely-linearization point inference (§5.2) relies on program dependence analysis. To track the persistent state (*i.e.*, whether an NVM address is persisted or not) for adversarial NVM state construction (§5.3), we instrument all flush and memory fence instructions. We also trace lock operations for adversarial thread interleaving construction (§5.3). For durable linearization validation (§5.4), we trace the value of each store instruction.

We implement an LLVM compiler pass [11] for the instrumentation and execute the instrumented binary with a test case to collect an execution trace. To ensure the total ordering in the execution trace for the analysis of multi-threaded programs, we protect our tracing code using a global mutex.

5.2 Likely-Linearization Point Inference

DURINN infers likely-linearization points by analyzing three concurrent NVM programming practices: *Atomic instruction*, *Guarded-Protection* and *Publish-after-Initialization*.

Atomic Instruction. In lock-free data structures, atomic instructions are typically used to update a synchronization variable and to make the effect of an operation atomically visible to other threads. Thus, DURINN identifies atomic instructions (*e.g.*, `CAS`, `fetch-and-add`) as likely-linearization points for lock-free writer operations (*e.g.*, `insert`).

Guarded-Protection. Guarded protection is a widely used NVM programming pattern (*e.g.*, key-value store, persistent data structure, file systems) to ensure atomic persistence of data [30]. A flag variable called *"guardian"* denotes whether the *"guarded data"* is valid or not. Thus, writing or reading a guardian is a linearization point. In Figure 8(a), for instance, a writer ensures that key and value are persisted before the `flag`, a guardian, is persisted. Also, a reader check if the `flag` (line 11) is set before reading the key and value (*"guarded read"*). Writing to the `flag` (line 7) is writer's linearization point since the changes become visible after setting the `flag`.

Based on this observation, DURINN performs program analysis to identify any stores to guardians. DURINN first finds out the guarded read pattern in the code to identify guardian candidates from conditional branch instructions. From the branch condition variables, DURINN performs the backward dataflow analysis to identify NVM memory addresses that are data-dependent on the branch condition variables. Then DURINN marks the stores to those NVM memory addresses as likely-linearization points.

Publish-after-Initialization. As an optimization to reduce persistence overhead, many NVM program follows so-called publish-after-initialization steps when adding a new memory object in the global data structure: (1) first allocating an NVM memory, (2) initializing the memory, and finally (3) linking (publishing) the memory to the global structure. For example, in Figure 8(b), a node is allocated first (line 2), then initialized (lines 4-10), finally is linked to the global list (`core→tail` at lines 13-15). The benefit of the publish-after-initialization idiom is that any writes to the new memory (lines 4-10) are not externally visible so that the persistence ordering of the writes in the initialization phase is relaxed until the new memory is published (line 13), improving performance (only one `fence` is needed at line 10).

Based on this NVM programming idiom, we filter out all the stores to newly allocated memory regions within an operation, and exclude them from likely-linearization points. We found that this pruning is highly useful for operations requiring many writes, such as node split/merge operations for a tree and a rehashing operation for a hash table.

[2]Non-temporal stores are supported/modeled as store+flush.

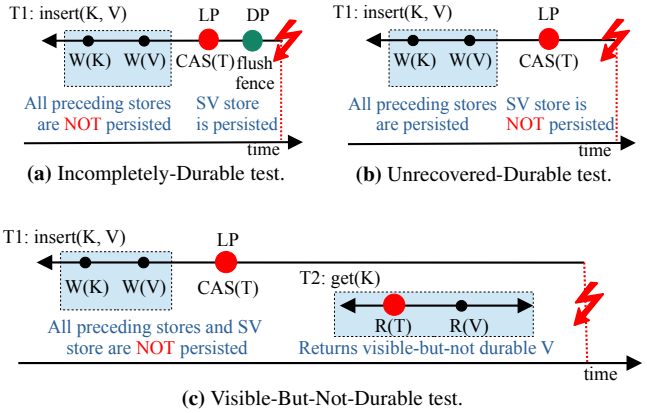


Figure 9: Adversarial test strategies for Incompletely-Durable, Unrecovered-Durable, and Visible-But-Not-Durable bugs. LP: linearization point. DP: durability point. SV: synchronization variable.

5.3 Adversarial NVM State and Thread Interleaving Construction

In this section, we first describe our adversarial construction approaches for each DL bug pattern (§5.3.1, §5.3.2, and §5.3.3). We then introduce our cache/NVM simulations to generate feasible NVM states (§5.3.4) for the validation.

5.3.1 Incompletely-Durable Bug Pattern

Testing Incompletely-Durable bugs can be performed for each operation in isolation without considering concurrent operations. When a crash happens after DP, to be durable linearizable, the crashing operation should provide the "all" (fully-executed) semantic and ensure that its effect remains visible after a crash (Figure 3). Then, the adversarial NVM state that increases the chance to trigger Incompletely-Durable bugs for a crash after DP would be to make all the preceding stores as unpersisted as possible. In other words, we artificially attempt to create a feasible yet worst NVM state that many updates made by an operation are not persisted.

Figure 9a illustrates our adversarial NVM state construction for `insert(K,V)` in which an atomic update to a synchronization variable `T` serves as LP and persisting it serves as DP. The adversarial NVM state would be to make the change to `T` persisted, but leave the changes to key and value unpersisted so that the new key and value data is not visible after a crash even though the synchronization variable `T` says differently. Note that we attempt to leave stores unpersisted only if possible. We do not force. We obey memory consistency and persistence model (*e.g.*, the semantics of fence, flush).

5.3.2 Unrecovered-Durable Bug Pattern

Testing Unrecovered-Durable bugs can also be performed for each operation in isolation. If a crash happens before DP, for durable linearizability, the crashing operation should provide the "nothing" (not-at-all-executed) semantic and ensure that any partial update is not visible after a crash (Figure 4). Then,

the adversarial NVM state that stress-tests the data structure under test to expose Unrecovered-Durable bugs for a crash before DP would be to make all the preceding stores as persisted as possible. That is, we are interested in constructing a feasible yet worst NVM state that many updates made by an operation are persisted, stress-testing its recovery logic.

Figure 9b shows our adversarial NVM state construction for the same `insert(K,V)` example in which `CAS(T)` is LP and persisting it is DP. We construct the adversarial NVM state such that the changes to key and value are persisted, but not the synchronization variable `T`. This way, the new key and value data may be visible after a crash when the synchronization variable `T` says they should not.

5.3.3 Visible-But-Not-Durable Bug Pattern

The Visible-But-Not-Durable bugs are related to the case where an operation takes an action after observing a visible-yet-not-durable state of another concurrent operation (Figure 5). Unlike the prior two bug patterns, testing Visible-But-Not-Durable bugs should be performed in a context sensitive manner. Figure 9c illustrates our adversarial NVM state and thread interleaving method for Visible-But-Not-Durable bugs, which requires the following three conditions.

Requirements. First, DURINN needs *(1) racy operations*. In Figure 9c, thread T1's `insert(K,V)` writes (`CAS`) on `T` and T2's `get(K)` reads `T`. Second, DURINN needs some *(2) prefix operations* (a sequence of other operations to execute before testing racy operations) that construct the preconditions for a race condition to be triggered. For example, an NVM data structure should be in a certain state (*e.g.*, initiating a resizing or node splitting process) to exhibit a race condition. Last, DURINN needs to control *(3) precise thread interleaving* in which a thread makes a progress based on another thread's visible-but-not-durable effect and a crash happens between LP and DP as illustrated in Figure 9c.

Challenges. However, constructing the test scenarios that satisfy all the three conditions is very challenging because not only search space is huge but also the three conditions are inter-dependent. For example, two racy operations with one sequence of prefix operations may not be racy any more with another sequence of prefix operations.

Our Approach. We propose techniques to find out adversarial (1) racy operations, (2) prefix operations, and (3) thread interleaving in a scalable manner by analyzing a *single-threaded* execution trace. Figure 10 shows the overall workflow. First, DURINN detects potentially racy two operation by analyzing a single-threaded memory trace. Second, if two racy operations are not consecutive, DURINN reorders the operations of the test case, places the two operations consecutively, and checks whether the same race can be triggered: *i.e.*, the new memory trace with the re-ordered operations still include the same race. Last, if two re-ordered operations are still racy, DURINN generates adversarial thread interleaving for these two operations. In the rest, we discuss each step in detail.

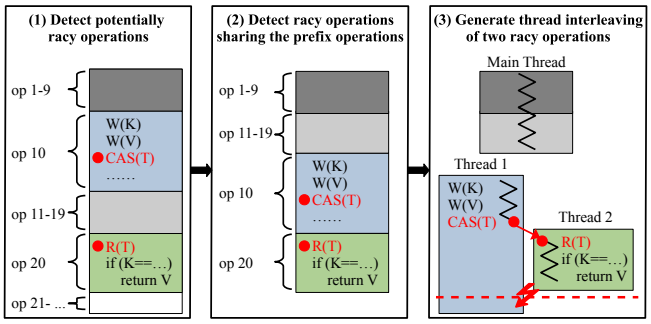


Figure 10: The workflow of adversarial NVM state and thread interleaving construction for Visible-But-Not-Durable bugs.

(1) Finding racy operations. The first step is to find potentially race operations. The inputs for the analysis are a single-threaded execution trace (§5.1) and the inferred likely-linearization points (§5.2). DURINN finds a pair of potentially racy operations that write-write or write-read synchronization variables (updated at likely-linearization points). These two potentially racy operations are not necessary to be consecutive in a single-threaded execution trace. In Figure 10 (1), operation 10 and 20 are such potentially racy operations.

(2) Finding prefix operations. A pair of potentially racy operations from the first step may not be racy when run in parallel. One main reason is that these two operations ran with different preceding operations (*i.e.*, prefix operations). In Figure 10 (1), the prefix of operation 10 is operation 1-9 but the prefix operations of operation 20 is operation 1-19. Hence, the precondition of an NVM data structure when running these two operations may be different, so these two operations may not be racy when run in parallel.

To filter out such spurious racy operations, DURINN reorders operations such that the two operations have the common prefix operations and places two potentially racy operations consecutively. In Figure 10 (2), operations 1-9 and 11-19 becomes the prefix of operation 10 and 20. We then run the instrumented program with the prefix and the two racy operations in a single thread and generate a new execution trace. If two candidates (operation 10 and 20) are still racy with the re-arranged operations, the prefix and racy operations will be fed in to the last step to construct thread interleaving of the two racy operations.

(3) Controlling thread interleaving and generating NVM state. For a given prefix operations, DURINN should precisely control thread interleaving of two concurrent racy operations. For example, in Figure 10 (3), thread 1 writes synchronization variable `T` first, which is LP, then thread 2 preempts and reads `T` then returns `V` to user. A crash should happen right after when the operation in thread 2 finishes and before thread 1 executes DP to trigger a Visible-But-Not-Durable bug.

In order to precisely control thread interleaving, DURINN uses a runtime technique using breakpoints. DURINN sets

breakpoints at the load and store of the synchronization variable (*e.g.*, `T` in Figure 10). After executing the prefix operations in a single-threaded manner, DURINN lets thread 1 run until reaching to the breakpoint of the `store` instruction to the synchronization variable. Then DURINN lets thread 2 run until it reaches the breakpoint of the `load` instruction of the synchronization variable. Then DURINN lets thread 2 resume and injects a crash right after finishing its operation.

Upon the crash, DURINN leaves all stores in thread 1 unpersisted as an adversarial NVM state. Note that thread 2 may not be able to finish its operation if other synchronization with thread 1 is involved (*e.g.*, deadlock). DURINN detects such a case with a timeout, and regards such thread interleaving infeasible. DURINN generates a `gdb` command script to automate the whole process. Using a hardware breakpoint makes DURINN's adversarial thread interleaving control efficent, though DURINN serializes a multi-threaded execution.

5.3.4 Cache and NVM Simulation

DURINN generates NVM crash images according to the adversarial NVM state and thread interleaving testing methods. To consider only feasible NVM states, DURINN simulates cache behaviors while obeying processor's memory consistency and persistence models. DURINN starts from the empty cache and NVM states and simulate the effects of `store`, `flush`, and `fence` instructions along an execution trace. Particularly, we implemented Intel's x86-64 architecture model following total store order (TSO) memory model consistency model [39, 56].

5.4 Durable Linearizability Validation

DURINN runs the NVM data structure under test from an NVM crash image (generated in §5.3) and checks if it violates durable linearizability by executing a sequence of validating operations. At a high level, the validating operations checks whether all operations before crash take effects (DL's C2 condition in §2.2); and whether the crashed operation is either fully executed or not at all executed (C3 condition). DURINN runs recovery code before running validation operations.

More specifically, for an NVM index data structure (*e.g.*, hash table, B-tree), the validating operations comprise:

1. A list of `get` operations to check all previously inserted but not deleted key-values exist,
2. A `get` operation to check the crash operations follows all or nothing semantics,
3. A list of `delete` operations for all inserted keys, and
4. A list of `get` operations to check all the deleted keys in the previous step are indeed deleted.

Note that for each crashed image, a list of completed operations and the crashing operation are known, so we know which key has been inserted or deleted. DURINN provide similar validating operations for other data structures: *e.g.*, array and queue.

6 Implementation

We implemented tracing and data flow analysis in LLVM [11]. We automatically generated `gdb` command files based on the locations of breakpoints. To control the progress of each thread in `gdb`, we set `scheduler-locking` on. Our LLVM-related code comprises around 1900 lines of C++ code. Other DURINN components are written in 2700 lines of Python code. Our current prototype supports an NVM program built on PMDK `libpmem` or `libpmemobj` libraries to create/load an NVM image from/to disk. To ensure the virtual address of the `mmap`-ed NVM heap are the same across different executions, we set `PMEM_MMAP_HINT` environment variable [38]. The DURINN prototype is available at https://github.com/cosmoss-jigu/durinn.

7 Discussion

7.1 False Negatives and False Positives in DURINN

DURINN may have false negatives (*i.e.*, missing bugs) for three reasons. First, DURINN is a trace-based dynamic tool that takes a test case as input. DURINN may miss DL bugs that did not appear in a trace.

Second, DURINN's likely-LP inference is based on heuristics and may miss true LPs in theory. Missing LP means no adversarial testing, so DURINN may miss DL bugs. However, the proposed heuristics are built on common NVM programming practices, namely Guarded-Protection and Publish-after-Initialization presented in §5.2. As a result, our empirical study (§8.4) shows that the inferred likely-LPs do not miss manually-identified (true/oracle) LPs and DURINN does not miss any DL bugs detected with the oracle LPs.

Last, DURINN performs adversarial testing and does not explore all possible NVM states and thread interleaving. In theory, for Incompletely-Durable and Unrecovered-Durable bugs, some more complex combinations of persisted and unpersisted stores may be required to trigger a DL bug. For Visible-But-Not-Durable bugs, more than two concurrent thread interleaving may be needed to expose a DL bug. However, our empirical study (§8.5) shows that DURINN detects all the bugs reported by the state-of-the-art Witcher [30] and indeed found more new bugs with significantly fewer tests.

On the other hand, for a given trace under test, DURINN does not have false positives as DURINN performs durable linearizability validation (§5.4). Any crash NVM image (constructed by adversarial testing) that violates durable linearizability is indeed a definite clue of a true DL bug (by definition). We note that multiple durable linearizability violations may stem from one root cause.

7.2 Persistent Cache

Intel architecture is expected to adopt eADR support (Extended Asynchronous DRAM Refresh) [37] that includes a cache into the persistent domain. For an eADR-enabled Intel architecture, there will be no gap between LP and DP because once the effect of a store reaches a cache, it is guaranteed to

be written back to the NVM.

We expect DURINN remain useful when eADR is available for the following three reasons. First, eADR is unlikely to be added to all product lines due to the high cost of battery size. In the current Intel platforms (Optane 200), eADR support is optional and requires an additional backup battery [36]. As eADR is not expected to be available in all machines, NVM programmers would need to write a code to support both eADR and non-eADR machines. Second, eADR is not the panacea if non-temporal stores (`ntstores`) are used. `Ntstores` place data in the store buffer and bypass caches, which is outside eADR persistent domain. For example, PMDK `pmemcpy()` prefers to use `movnstore` for better performance. An NVM program may lead to an inconsistent state when data in a store buffer is not flushed into NVM before a crash. Last, the Unrecovered-Durable bug pattern is still an issue even with eADR because it requires recovering or tolerating any partial updates made before linearization point. eADR has nothing to do with such a recovery logic. Developers still need to design and implement inconsistency-recoverable data structures.

7.3 Relationship to ACID

Three DL bug patterns can be disucssed using traditional database/filesystem's ACID terms. One can view *Incompletely-Durable* and *Unrecovered-Durable* bugs as ACID-atomicity/consistency/durability violations *Incompletely-Durable* bug pattern considers a crash after DP and tests ACID-atomicity/consistency/durability's fully-executed "all" semantic. *Unrecovered-Durable* bug pattern considers a crash before DP and tests ACID-atomicity's not-at-all-executed "nothing" semantic.

On the other hand, *Visible-But-Not-Durable* bug is related to ACID-isolation violation. *Visible-But-Not-Durable* bug pattern considers a crash before DP. However, a concurrent operation observed unpersisted data, and it completed, violating ACID-isolation and forcing the crashed operation to ensure the "all" semantic. A naive durability checker cannot detect *Visible-But-Not-Durable* bugs because they require a completed, ACID-isolation-violating, concurrent operation.

8 Evaluation

For evaluation, we first present our methodology (§8.1), then present the following experimental results.

- We report and analyze the DL bugs detected by DURINN (§8.2) along with detailed statistics, including the number of tests and testing time (§8.3).
- We evaluate the effectiveness and (empirical) soundness of DURINN's likely-linearization inference technique (§8.4).
- We compare DURINN with other NVM crash-consistency testing tools in terms of bug detection effectiveness and test space reduction (§8.5).

Application	Version	Type	Concurrency	Persistence
P-LF-BST [28]	5fa1dee	binary search tree	lock-free	LL
P-LF-Hash [28]	5fa1dee	hash table	lock-free	LL
P-LF-List [28]	5fa1dee	linked list	lock-free	LL
P-LF-Skiplist [28]	5fa1dee	skiplist	lock-free	LL
P-LF-Queue [29]	08fecfb	queue	lock-free	LL
CCEH [50]	d53b336	hash table	lock-based	LL
Fast Fair [34]	c86f5fb	B+ tree	lock-based	LL
P-ART [44]	5b4cf3e	radix tree	lock-based	LL
P-CLHT [44]	5b4cf3e	hash table	lock-based	LL
P-Hot [44]	5b4cf3e	trie	lock-based	LL
P-Masstree [44]	5b4cf3e	B tree + trie	lock-based	LL
pmdk-array [5]	v1.8	array	lock-based	LL
pmdk-queue [6]	v1.8	queue	lock-based	TX

LL: low-level persistence primitives **TX**: transactional persistence

Table 1: Tested concurrent NVM data structures

8.1 Evaluation Methodology

Tested NVM data structures. We evaluate DURINN with 13 concurrent NVM data structures, as listed in Table 1 with tested version, data structure type, its concurrency control mechanism, and persistence programming model. There are two concurrency control mechanisms: lock-free and lock-based. For persistence (durability) control, most data structures use low-level (LL) persistence primitives such as `flush` and `fence` instructions, while `pmdk-queue` uses PMDK's transactional (TX) persistence programming model.

All the tested data structures have been highly optimized for NVM, and most of them have shown to be more scalable than (simple) NVM hash tables and B-trees used in NVM-backed key-value stores such as `memcached`, `redis`, `pmemkv`, etc. All tested data structures use `libpmemobj`, the PMDK library for persistent memory allocation or transaction. As some data structures originally used a volatile memory allocator and emulated NVM using DRAM, we modified them to use the PMDK's persistent NVM memory allocator. Our changes do not add or delete any new/existing persistence primitives, or memory operations. Thus, the changes do not affect the bug detection evaluation.

Test Cases. We use AFL++ fuzzer [1] to generate a test case for our evaluation. We first feed a randomly generated seed into ALF++ fuzzer. Our random seed generator assigns a higher probability to create a new unused key for `insert`; and to reuse existing keys for other dependent operations such as `delete`, `update`, `query`. Then we run the fuzzer and picked the generated test case with the highest code coverage, which consists of 1,000 operations. We found that 1,000 operations are large enough to achieve a reasonable and stable code coverage (50%-80%) for our tested NVM data structures. Missing code coverage is due to unused features (*e.g.*, garbage collection) and debugging codes. The generated test cases are used for both likely-LP inference and adversarial testing.

Experimental setup. We ran all experiments on a 64-bit Fedora 29 machine with two 16-core Intel Xeon Gold 5218 processors (2.30GHz), 192 GB DRAM, and 512 GB NVM.

Name (Total #Bugs)	Bug ID	New	Confirm	Code	Type	Description	Impact	Fix strategy
P-LF-BST (1)	1	✓	✓	BSTAravindTraverse.h:331	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
P-LF-Hash (1)	2	✓	✓	ListTraverse.h:212	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
P-LF-List (1)	3	✓	✓	ListTraverse.h:212	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
P-LF-Skiplist(1)	4	✓	✓	SkiplistTraverse.h:218	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
P-LF-Queue(1)	5	✓	✓	DurableQueue.h:L74	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
CCEH (2)	6	✓		CCEH_MSB.cpp:280	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	7		✓	CCEH_MSB.cpp:103	DL2	Atomicity in rehashing	Unable to recover	inconsistency-recoverable design
FAST-FAIR (5)	8	✓	✓	btree.h:955,979	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	9	✓	✓	btree.h:955,1007	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	10		✓	btree.h:224	DL1	Missing persistence primitives	Lost key-value	add persistence primitives
	11		✓	btree.h:213	DL2	Partial inconsistency is never recovered	unable to recover	inconsistency-recoverable design
	12		✓	btree.h:576	DL2	Atomicity in node splitting	unable to recover	logging/transaction
P-ART (4)	13	✓		Tree.cpp:35,258	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	14	✓		Tree.cpp:35,384	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	15		✓	N16.cpp:15	DL2	Atomicity between metadata and key-value	Unable to recover	inconsistency-tolerable design [8]
	16		✓	N4.cpp:17	DL2	Atomicity between metadata and key-value	Unable to recover	inconsistency-tolerable design [8]
P-CLHT (3)	17	✓		clht_lb_res.c:315,370	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	18	✓		clht_lb_res.c:315,468	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	19		✓	clht_lb_res.c:166	DL1	Missing persistence primitives	Lost key-value	add persistence primitives [9]
P-HOT (4)	20	✓		HOTRowex.hpp:61,84	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	21		✓	TwoEntriesNode.hpp:30	DL1	Missing persistence primitives	Points to garbage	add persistence primitives [9]
	22		✓	HOTRowexNode.hpp:315	DL1	Missing persistence primitives	Points to garbage	add persistence primitives [9]
	23		✓	HOTRowex.hpp:270	DL1	Missing persistence primitives	Points to garbage	add persistence primitives [9]
P-Masstree (3)	24	✓		masstree.h:1837,744	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	25	✓		masstree.h:1837,941	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	26		✓	masstree.h:1378	DL2	Atomicity in node splitting	Unable to recover	logging/transaction
pmdk-array (1)	27		✓	array.c:486	DL2	Atomicity between metadata and data	Unable to recover	logging/transaxtion

DL1: Incompletely-Durable **DL2**: Unrecovered-Durable **DL3**: Visible-But-Not-Durable

Table 2: List of Durable Linearizability bugs detected by DURINN. In total, 27 (15 new) durable linearizability bugs were detected from 12 NVM data structures. There were 10 Incompletely-Durable bugs, 7 Unrecovered-Durable bugs, and 10 Visible-But-Not-Durable bugs.

App	# stores	# LPs	# DL1 tests	# DL2 tests	# DL3 tests	Execution time
P-LF-BST	10086	656	656	656	46	1m26s
P-LF-Hash	4604	547	547	547	5	1m44s
P-LF-List	4604	547	547	547	1623	7m15s
P-LF-Skiplist	26692	1040	1040	1040	491	4m3s
P-LF-Queue	9710	2000	2000	2000	7155	39m45s
CCEH	3631	1280	1280	1280	37	1m36s
Fast Fair	12989	10599	10599	10599	1585	8m37s
P-ART	12553	1112	1112	1112	287	2m34s
P-CLHT	2885	711	711	711	55	2m6s
P-HOT	32600	640	640	640	420	3m35s
P-Masstree	1403	1058	1058	1058	984	4m58s
pmdk-array	20505	3097	3097	3097	0	4m14s
pmdk-queue	57000	3000	3000	3000	0	2m51s
Total	199262	26287	26287	26287	12688	1h23m18s

DL1: Incompletely-Durable **DL2**: Unrecovered-Durable
DL3: Visible-But-Not-Durable

Table 3: The detailed statistics of DURINN bug finding.

8.2 Detected Durable Linearizability Bugs

In summary, DURINN detected 27 (15 new) durable linearizability bugs from 12 NVM data structures. There were 10 Incompletely-Durable bugs, 7 Unrecovered-Durable bugs and 10 Visible-But-Not-Durable bugs. 7 out of 15 new bugs have been confirmed by the developers so far. Table 2 shows the source code locations, impacts and fix strategies of the detected bugs.

(DL1) Incompletely-Durable bugs. DURINN detected 10 Incompletely-Durable bugs. Figure 6(a) discussed in §3.2 is a representative example (Bug ID 19) found in P-CLHT, leading to a lost key-value. As another instance, in P-LF-List (Bug ID 3), a new node is not fully persisted before it is added to the list using a `CAS` operation (which is LP). If a crash happens before DP (and after LP in this particular case), the list may contain a garbage node leading to an inconsistent structure. To fix Incompletely-Durable bugs, developers need to persist all the changes using additional cache line flush and fence instructions before DP.

(DL2) Unrecovered-Durable bugs. DURINN detected 7 Unrecovered-Durable bugs. Figure 6(b) illustrates a case detected in Fast-Fair (Bug ID 12). For another example, in CCEH (Bug ID 7), if a crash happens while rehashing the table and before adding a new segment into the table, the hash table will be in an illegal state: *i.e.*, all the metadata assumes there is a new segment added but it is not. To fix Unrecovered-Durable bugs, an NVM data structure should be able to recover from or tolerate partial updates before LP of an operation. Designing an inconsistency-recoverable design is one solution. Using logging or transaction is another.

(DL3) Visible-But-Not-Durable bugs. DURINN detected 10 Visible-But-Not-Durable bugs. Figure 6(c) shows a Visible-But-Not-Durable bug in Fast-Fair (Bug ID 8). For another example, Bug ID 6 from CCEH is due to incorrect usage of locks. While both `insert` and `get` operations use a lock to protect a critical section, the write to the synchronization variable (LP) is inside the critical section but the persistence of the synchronization variable (DP) is ensured outside the critical section in `insert`. Since the DP is not protected by a lock, the `get` operation is able to observe the visible but not durable writes from a concurrent `insert` operation. We observed two ways to fix Visible-But-Not-Durable bugs. Some

choose to fix the concurrency control mechanism to guarantee that every data read by concurrent threads is persisted. Others made one operation that reads unpersisted data help persist the data on behalf of another concurrent operation.

8.3 Statistics of DL Bug Detection

Table 3 shows the detailed statistics of DURINN when tested with 1000 operations. The second column reports the number of stores and the third column lists the number of inferred likely linearization points. On average, using static analysis described in §5.2, DURINN infers about 2,000 likely-LPs, which is 13% of 15.3K NVM stores traced while running 1000 tested operations. More detailed analysis on likely-LP inference will follow in §8.4.

The next three columns show the number of DL tests performed by DURINN to detect three DL bug patterns. The number of Incompletely-Durable and Unrecovered-Durable tests are the same as the number of inferred LPs because for each LP, DURINN performs one adversarial test for Incompletely-Durable bugs and for Unrecovered-Durable bugs. On the other hand, the number of Visible-But-Not-Durable tests depends on the number of co-schedulable racy operations. The second last column shows that the number of Visible-But-Not-Durable tests varies by data structures up to a few thousand. Intuitively, lock-free data structures tend to have more co-schedulable racy operations than (coarse-grained) lock-based ones, requiring more tests. The last column reports the execution time, which mostly depends on the number of tests. Testing all three test cases typically takes a few minutes. `P-LF-Queue` took the most time (around 40 mins) due to the large number of concurrent Visible-But-Not-Durable testing.

Lastly, for each test case violating durable linearizability, we manually analyze each case and report the details in Table 2. DURINN provides sufficient information for root cause analysis, including execution trace, crash location, persisted and unpersisted writes, and a crash NVM image. We loaded the crash image in `gdb` and followed the DURINN-generated schedule to inspect the root causes of detected DL bugs.

8.4 Likely-Linearization Point Inference

DURINN infers likely-linearization points using *Guarded-Protection* and *Publish-after-Initialization* heuristics described in §5.2. The number of likely-LP determines the number of DL tests that DURINN performs, so in terms of scalability, the less the better. At the same time, ideally, likely-LPs should not miss true LPs because missing LPs may lead to missing true DL bugs (false negatives).

We performed a detailed case study with CCEH and Fast-Fair in which we manually analyzed the true LPs (oracle) for comparison. They both use lock-based concurrency control in which the store instructions serving as LPs are not explicit. They are non-trivial concurrent data structures including balancing operations such as rehashing (CCEH) and node split/merge (Fast-Fair) operations.

Figure 11 shows the effectiveness of the proposed likely-LP inference techniques, compared to the manually identified LPs. The first bar represents the number of total stores. The second and third bar represent the number of likely-LPs when only Guarded-Protection or Publish-after-Initialization heuristics is used, respectively. The fourth bar shows the number of likely-LPs of DURINN where both are considered. The last bar is the number of LPs from our manual source code analysis. The result shows that DURINN effectively reduces the number of likely-LPs using two heuristics. The number of likely-LPs inferred by DURINN is twice as the number of manually-identified LPs. Note that as listed in Table 3, CCEH and Fast Fair are the most difficult data structures in terms of the reduction ratio between the stores and the likely-LPs.

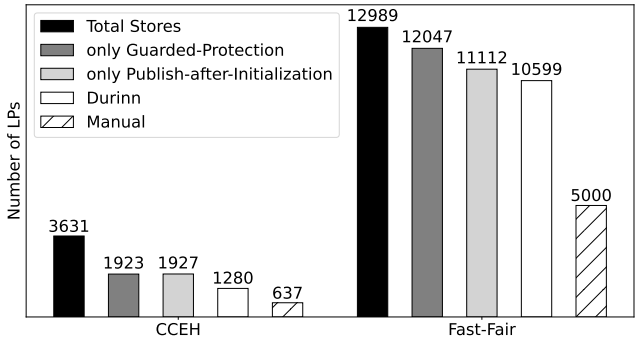


Figure 11: A case study of likely-linearization point inference.

Additionally, we compared the bug detection effectiveness and found that DURINN's inferred likely-LPs detect the same DL bugs as manually-identified (true) LPs. Though DURINN's likely-LP inference heuristics do not guarantee soundness in theory, this experiment empirically shows that likely-LP inference did not miss true LPs (at least) for the CCEH and Fast-Fair. We believe the same case for other data structures given that the heuristics are designed based on common NVM programming patterns.

8.5 Comparison with Other Tools

We present the detailed comparison with Witcher [30], the state-of-the-art NVM crash-consistency bug detector, and Yat [43], an exhaustive crash-consistency testing tool.

Bug Detection. We compared the bug detection effectiveness with Witcher. In their paper, Witcher claims that it can detect all the crash-consistency bugs that prior tools (*e.g.*, PMTest, XFDetector and Agamotto) found for a common set of NVM programs, along with some new bugs. For comparison, we run Witcher with the same test case with 1000 operations for six common data structures: CCEH, Fast-Fair, P-ART, P-CLHT, P-HOT and P-Masstree. Both Witcher and DURINN detected 11 bugs in common. Beyond them, DURINN reports 10 Visible-But-Not-Durable bugs that Witcher missed. Detecting Visible-But-Not-Durable bugs requires scheduling concurrent operations, which is not supported by Witcher.

Test Space Reduction. We compare the number of tests

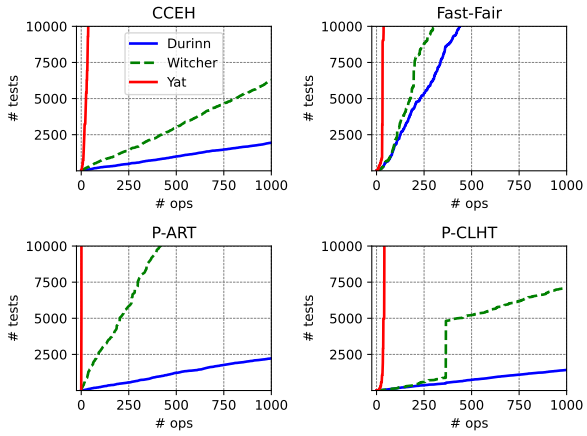


Figure 12: Test space comparison.

performed by DURINN, Yat, and Witcher over the same 1000 operations. Figure 12 shows how the number of tests grows (y-axis) as the number of tested operations increases (x-axis) for four data structures: CCEH, Fast Fair, P-ART, and P-CLHT. The other NVM data structures demonstrate a similar pattern. Yat is an exhaustive testing tool, so the test space explodes within the first ten operations. Witcher performs several times more tests than DURINN. The sudden spike in P-CLHT is due to a rehashing operation. On the other hand, DURINN performs adversarial testing for three DL bug patterns, reduces the number of tests, yet still detects more bugs than Witcher.

9 Related Work

File system/database Consistency Checking. File system consistency checking [17, 18, 32, 41, 48, 49, 54, 57, 60–62] deals with block-grained files, and logging/journaling is the norm for crash consistency. On the other hand, DURINN deals with concurrent data structures backed by byte-addressable NVM. DURINN analyzes load/store instructions along with cacheline flush and fence instructions, controlling durability in NVM. NVM data structures often come with custom (log-free) crash consistency and lock-free logic, making NVM test space huge. For file systems, CrashMonkey [49] bounds search space using heuristics learned from a bug study. EXPLODE [61] and FiSC [62] use in-situ model checking. In contrast, DURINN reduces test space with adversarial crash state and thread interleaving construction.

For correctness conditions, strict serializability and durable linearizability are equivalent from the ACID properties perspective. (PMDK "transaction" does not provide "isolation", though). Yet, we believe durable linearizability is the appropriate framework to use as many NVM data structures are derived from volatile concurrent data structures, where linearizability is the norm. Others [28, 42] also use durable linearizability. Equivalently, the asynchronous commit mechanism in database systems (Salt [59] and Hekaton [23]) can be mapped to "buffered durable linearizability" [40]. For performance, both do not eagerly make the changes durable as long as they can resume from one of the old consistent states.

Linearizability Checker. Line-up [15] is the first complete and automatic checker for deterministic linearizability. It detects thread-safety violations by comparing the concurrent execution to linearizable executions of a test. Similarly, Round-up [63] checks quasi linearizability. Quasi linearizability intentionally introduces non-determinism into the parallel computations and exploits such non-determinism to improve the performance. Pradel *et al.* [53] detects concurrency bugs in thread-safe classes. It generates tests in which multiple threads call methods on a shared instance of the tested class and check if the execution matches any linearizable execution.

Bug Detector for NVM Software. Most existing NVM bug detectors are not designed for durable linearizability bugs. PMDebugger [22] targets universal bugs such as missing persistence primitives. To detect application-specific bugs, such as persistence ordering/atomicity bugs, Yat [43], PMTest [46], XFDetector [45] and Agamotto [51] require user-defined custom oracles or consistency checkers. Jaaru [31] only identifies bugs that have visible manifestation, such as a segment fault or an assertion failure. Witcher [30] leverages all or nothing semantics for validation like DURINN, but it is limited to single-threaded NVM programs.

Active Testing. AtomFuzzer [52] is a randomized active atomicity violation detector, which modifies the thread scheduler behavior to create atomicity violations with high probability. RaceFuzzer [55] uses potential data race information obtained from an existing dynamic analysis technique to control a random scheduler of threads for actively detecting race conditions. Jumble [27] uses adversarial memory to classify race conditions as destructive or benign on systems with relaxed memory models. Relaxer [16] detects sequential consistency violations in a relaxed memory model by actively leading execution to predicted violations.

10 Conclusion

We present DURINN, the first durable linearizability checker for concurrent NVM data structures. We explore the gap between linearizability point and durability point, and define three novel durable linearizability bug patterns. We propose adversarial crash state and thread interleaving construction and likely-linearization point inference to detect durable linearizability bugs in an active and scalable manner. DURINN detected 27 (15 new) durable linearizability bugs.

Acknowledgments

We thank the anonymous reviewers and Murat Demirbas (our shepherd) for their insightful comments and feedback. This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant from the Korean government (MSIT) (No. 2014-3-00035) and by the National Science Foundation under the grants CNS-2029720 and CCF-2153747.

References

[1] American Fuzzy Lop plus plus (AFL++). URL: https://github.com/AFLplusplus/AFLplusplus.

[2] Argonne National Lab's Aurora Exascale System. URL: https://www.intel.com/content/www/us/en/customer-spotlight/stories/argonne-aurora-customer-story.html.

[3] Available first on Google Cloud: Intel Optane DC Persistent Memory. URL: https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud_intel-optane-dc-persistent-memory.

[4] Key/Value Datastore for Persistent Memory. URL: https://github.com/pmem/pmemkv.

[5] Persistent array in PMDK. URL: https://github.com/pmem/pmdk/tree/stable-1.8/src/examples/libpmemobj/array.

[6] Persistent queue in PMDK. URL: https://github.com/pmem/pmdk/tree/stable-1.8/src/examples/libpmemobj/queue.

[7] Pmem-Memcached. https://github.com/lenovo/memcached-pmem.

[8] RECIPE commit to fix reported bugs. URL: https://github.com/utsaslab/RECIPE/commit/4b0c27674ca7727195152b5604d71f47c0a0a7a2.

[9] RECIPE commit to fix reported bugs. URL: https://github.com/utsaslab/RECIPE/commit/950ae0ea5ed23ce28840615976e03338b943d57a.

[10] Redis v3.2. https://github.com/pmem/redis/tree/3.2-nvml.

[11] The LLVM Compiler Infrastructure. URL: https://llvm.org/.

[12] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. Bbb: Simplifying persistent programming using battery-backed buffers. In *Proceedings of the 27rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 111–124, Seoul, South Korea, February 2021.

[13] Anandtech. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here!, 2018. URL: https://www.anandtech.com/show/12828/intel-launches-optane-dimms_up-to-512gb-apache-pass-is-here.

[14] ARM Limited. ARM architecture reference manual armv8, for armv8-a architecture profile, 2020.

[15] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 330–340, New York, NY, USA, 2010. Association for Computing Machinery. URL: https://doi.org/10.1145/1806596.1806634, doi:10.1145/1806596.1806634.

[16] Jacob Burnim, Koushik Sen, and Christos Stergiou. Testing concurrent programs on relaxed memory models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 122–132, Toronto, Canada, July 2011.

[17] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 270–286, 2017.

[18] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, 2015.

[19] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[20] CXL Consortium. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.

[21] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[22] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible and comprehensive bug detection for persistent memory programs extended abstract. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

[23] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD/PODS Conference*, pages 1243–1254, New York, USA, June 2013. ACM.

[24] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 478–493, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3341301.3359637, doi:10.1145/3341301.3359637.

[25] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[26] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, April 2014.

[27] Cormac Flanagan and Stephen N Freund. Adversarial memory for detecting destructive races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 244–254, Toronto, Canada, June 2010.

[28] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. NVTraverse: In NVRAM Data Structures, the Destination Is More Important Than the Journey. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 377–392, Virtual, June 2020.

[29] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, wien, Austria, March 2018.

[30] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 100–115, Virtual, October 2021.

[31] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

[32] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 14:1–14:16, 2008.

[33] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. In *Proceedings of the ACM Transactions on Programming Languages and Systems*, pages 463–492, 1990.

[34] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 187–200, Oakland, California, USA, February 2018.

[35] Intel. pmreorder, 2019. URL: https://pmem.io/pmdk/manpages/linux/master/pmreorder/pmreorder.1.html.

[36] INTEL. Third Generation Intel® Xeon® Processor Scalable Family Technical Overview, 2020. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-xeon-processor-scalable-family-overview.html.

[37] Intel. eADR: New Opportunities for Persistent Memory Applications, 2021. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent_memory-applications.html.

[38] INTEL. PMDK man page: libpmem - persistent memory support library, 2021. URL: https://pmem.io/pmdk/manpages/linux/v1.0/libpmem.3.html.

[39] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2021. https://software.intel.com/en-us/articles/intel-sdm.

[40] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the 30st International Conference on Distributed Computing (DISC)*, pages 313–327, Paris, France, September 2016.

[41] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 147–161, 2019.

[42] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *SOSP '21: 28th ACM Symposium on Operating Systems Principles, October 25-28, 2021*. ACM, 2021.

[43] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, Philadelphia, PA, June 2014.

[44] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[45] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 1187–1202, Lausanne, Switzerland, April 2020.

[46] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 411–425, Providence, RI, April 2019.

[47] Micro. 3D XPoint Technology, 2019. URL: https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[48] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377, 2015.

[49] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 33–50, Carlsbad, CA, October 2018.

[50] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, Boston, MA, February 2019.

[51] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/neal.

[52] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 135–145, Atlanta, GA, November 2008.

[53] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. *SIGPLAN Not.*, 47(6):521–530, June 2012. URL: https://doi.org/10.1145/2345156.2254126, doi:10.1145/2345156.2254126.

[54] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–280, Dublin, Ireland, June 2009.

[55] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21, Tucson, AZ, June 2008.

[56] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[57] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2016.

[58] Steve Scargall. Programming Persistent Memory: A Comprehensive Guide for Developers, 2020. https://pmem.io/book/.

[59] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining acid and base in a distributed database. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[60] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[61] Junfeng Yang, Can Sar, and Dawson Engler. explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 10–10, Seattle, WA, November 2006.

[62] Junfeng Yang, Paul Twohey, and Dawson. Using model checking to find serious file system errors. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–288, San Francisco, CA, December 2004.

[63] Lu Zhang, Arijit Chattopadhyay, and Chao Wang. Round-up: Runtime checking quasi linearizability of concurrent data structures. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, page 4–14. IEEE Press, 2013. URL: https://doi.org/10.1109/ASE.2013.6693061, doi:10.1109/ASE.2013.6693061.

SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute

Ningxin Zheng[1*], Bin Lin[1,2*], Quanlu Zhang[1], Lingxiao Ma[1], Yuqing Yang[1], Fan Yang[1], Yang Wang[1], Mao Yang[1], Lidong Zhou[1]

[1]Microsoft Research, [2]Tsinghua University

Abstract

Sparsity is becoming arguably the most critical dimension to explore for efficiency and scalability, as deep learning models grow significantly larger and more complex. After all, the biological neural networks, where deep learning draws inspirations, are naturally sparse and highly efficient.

We advocate an end-to-end approach to model sparsity via a new abstraction called Tensor-with-Sparsity-Attribute (*TeSA*), which augments the default Tensor abstraction that is fundamentally designed for dense models. TeSA enables the sparsity attributes and patterns (*e.g.,* for pruning and quantization) to be specified, propagated forward and backward across the entire deep learning model, and used to create highly efficient, specialized operators, taking into account the execution efficiency of different sparsity patterns on different (sparsity-aware) hardware. The resulting SparTA framework can accommodate various sparsity patterns and optimization techniques, delivering 1.7x∼8.4x average speedup on inference latency compared to seven state-of-the-art (sparse) solutions with smaller memory footprints. As an end-to-end model sparsity framework, SparTA facilitates sparsity algorithms to explore better sparse models.

1 Introduction

As deep neural network (DNN) models become large and complex, they are inevitably getting sparse (or made sparse) for efficiency, just as manifested in the highly sparse biological neural networks [89]. A DNN model is usually modeled as a data flow graph (DFG), where each node is an operator with one or multiple input and output tensors. *Model sparsity* involves introducing some sparsity patterns on the tensors; for example, to quantize some tensors with lower precision (*e.g.,* 16 to 8-bit); to prune the model by setting the value of some (or all) parts of some tensors to zero (*e.g.,* block sparsity [61, 63] or fine-grained sparsity [43, 54, 55]); or to apply the combination of pruning and quantization to a model. With careful pruning and quantization, a DNN model can be compressed into a smaller memory footprint without losing too much accuracy. With DNN operators customized for the sparsity patterns, the resulting model will, hopefully, come with a lower inference latency.

Unfortunately, deep learning systems are not yet effective in exploiting sparsity: the increase in sparsity might not translate into actual gains in efficiency for a variety of reasons. First, the computation kernels for general sparse operations remain far from optimal. For example, cuSPARSE [3], the CUDA library for sparse matrix operations, has been shown to underperform cuBLAS, its dense counterpart, even when the sparsity of the matrices reaches 98% (Table 1). Second, as DNN computation usually takes multiple stages, the sparsity pattern might vary significantly across stages, making it hard to develop sparsity-aware optimizations for end-to-end gains. Finally, any effective sparsity-aware optimization might involve additional support across the vertical stack, from the deep learning framework, compiler, optimizer, operators and kernels, and all the way to hardware. Insufficient support at any of the layers could lead to inefficiency.

We therefore propose SparTA, a new framework that treats sparsity as a first-class citizen, with the following design principles. The design is *customizable and extensible* to accommodate new innovations on model sparsity; it is *end-to-end* and covers the *whole-stack*, rather than being limited to one operator or to one layer; it aims for *extreme performance* without sacrificing general applicability; it can facilitate existing sparsity algorithms to explore sparse models more efficiently.

At the core of SparTA is a new abstraction, *Tensor-with-Sparsity-Attribute* or *TeSA*, which augments the standard tensors with attributes to describe sparsity properties and patterns. Examples include low-precision weights, block (structured) sparsity, and fine-grained (unstructured) sparsity. A set of *TeSA propagation rules* guides the forward and backward propagation of sparsity attributes for end-to-end coverage. The rules can either be defined by the proposed TeSA algebra,

*: Equal contribution.

Table 1: Speed of matrix multiplication ($1024 \times 1024 \times 1024$) in cuSPARSE and cuBLAS on NVIDIA 2080Ti (unit: us).

Sparsity Ratio	50%	90%	95%	99%
cuSPARSE	1652.5	633.9	463.0	181.7
cuBLAS	208.3	208.3	208.3	208.3

or be inferred in a probabilistic way (§3.2).

With the sparse attributes in TeSA, SparTA can generate an efficient execution plan, taking into account the sparsity-aware hardware and specific sparse operators/kernels in certain sparsity patterns and conditions. SparTA may transform an execution plan to decompose complex sparsity attributes into a combination of simple ones with known effective optimizations. In the execution plan, SparTA can perform *code specialization* to generate efficient kernels for simple and regular sparse attributes, instead of resorting to generic but less efficient sparse kernels. This is how SparTA achieves extreme efficiency without sacrificing generality (§3.3).

Due to the whole-stack support (all the way to the codegen on accelerators), SparTA is able to provide the ground-truth performance metrics (*e.g.,* latency) that can help evaluate different execution plans given a TeSA with fixed sparsity attributes and also offer valuable feedback for practitioners to search for the set of sparsity attributes with the ideal tradeoff between performance and accuracy (§5.4).

SparTA is highly customizable and extensible. With TeSA, one can define new sparsity properties and patterns for new ways of exploiting sparsity, provide new TeSA propagation rules, and incorporate new sparsity-aware operators, kernels, and (sparsity-aware) hardware accelerators.

We have implemented SparTA based on Rammer [60], a state-of-the-art open-source DNN compiler with no special support for sparsity. We extensively evaluate SparTA on three popular DNN models with four representative sparsity patterns on three accelerators (*i.e.,* CUDA GPU, ROCm GPU, Intel CPU). Our evaluation shows that SparTA achieves up to 8.4x average speedup on model inference latency with less memory consumption, compared to seven state-of-the-art solutions (§5). We have also used SparTA to speed up sparse DNN model training and achieved more than 2x speedup than previous solutions (§5.5). By open sourcing SparTA[1], we hope that this work can bring the community together in this extensible and unified framework to accelerate innovations on model sparsity.

2 Background and Motivation

The size of deep neural networks grows significantly over the past years [25, 37], which incurs large inference latency and heavy memory burden. Model sparsity is arguably the most critical dimension to explore for efficiency and scalability.

[1]Code available at https://github.com/microsoft/SparTA

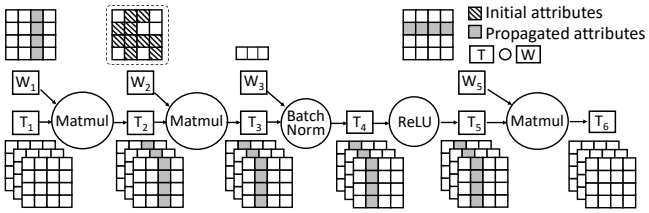


Figure 1: The sparsity attribute of one tensor can be propagated along the deep learning network.

Various forms of sparsity. Deep learning model sparsity is an active and extensively studied research topic. Currently, there are various sparsity patterns being studied. Structured (coarse-grained) sparsity, including channel-granularity sparsity, and block sparsity [56, 59, 61, 63], involves pruning a channel or a sub-block of tensors (e.g., weight or activation tensor) associated with some operators. With unstructured (fine-grained) sparsity, any element of a tensor [43, 54, 55] might be pruned. Quantization algorithms represent models at different levels of precision (*e.g.,* binarized models [31], 8-bit models [52, 92]), and even with different, mixed precision across neural network layers [36, 57, 77] or within a single tensor [66, 84]. Some research further combines pruning and quantization in order to achieve high accuracy under the strict latency and memory constraints [42, 74, 75, 78, 83, 90]. Overall, pruning and quantization have been shown effective in reducing the size and computation complexity of certain deep learning models, sometimes by more than 10 times, without losing much accuracy [42, 76].

The myth of FLOPs. Model sparsity does not translate directly into performance benefits. The existing practice of using "proxy metric" (*e.g.,* FLOPs, or Floating point operations) to evaluate the effect of their proposals such as model inference latency is flawed and leads to inaccurate results. For example, when an operator's weight is pruned by 50% with fine-grained sparsity, even though in theory its FLOPs can be reduced by half, the actual inference latency may become higher with a default sparse kernel (§5.3).

One reason is the sub-optimal implementation of current *generic* sparse kernels. A generic sparse kernel tends to apply a few default sparse encoding schemes (*e.g.,* Compressed Sparse Row [26]) to any sparse tensors. This may miss optimization opportunities in a tensor with a specific sparsity pattern, such as structured sparsity. As a result, a generic sparse kernel library like cuSPARSE [3] can outperform cuBLAS, its dense counterpart [2], only in some extreme sparse case (98%), as shown in Table 1. This motivates the need to find a general framework to implement *specialized* kernels tailored for individual sparsity schemes.

The diminishing end-to-end returns. Sparsity algorithms often focus on exploring the sparsity of a certain DNN operator (*e.g.,* convolution [64]). However, when placed in an end-to-end deep learning model, the sparsity pattern across the whole model can be impacted by each of the operators in the model. This may introduce sophisticated sparsity pat-

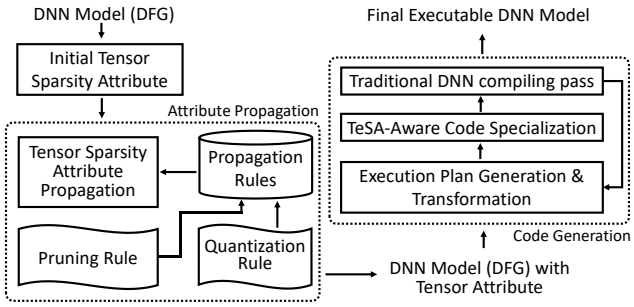


Figure 2: The system architecture of SparTA.

terns that are difficult for existing solutions to understand or optimize, leading to diminishing end-to-end return from sparsity.

In Figure 1, tensor W_2 illustrates a fine-grained sparsity pattern (63% sparsity). Such an initial sparsity pattern of W_2 incurs ripple effects. W_2 would propagate its sparsity attribute to the down-stream and up-stream tensors, including W_1, T_2, T_3, T_4, T_5, and W_5. For example, because the second column of W_2 is pruned, the second column of T_3 is destined to be all zero, hence can be pruned too (as $T_2 \times W_2 = T_3$). Likewise, as the third row of W_2 is pruned, the third column of T_2 can also be pruned. It is therefore desirable for a deep learning compiler to understand such *propagation* of sparsity so as for further sparsity-aware optimization.

Across-stack sparsity innovations in silos. Due to the above limitations, sparsity innovations either are constrained to individual operators and evaluated with proxy metrics without knowing the end-to-end effects, or have to be implemented manually on a few neural models, difficult to be ported to other models [42, 77]. More problematically, individual solutions are hard to be extended to or combined with other proposals. All these motivate SparTA, a common foundation to facilitate sparsity innovations, which can be evaluated end-to-end.

3 SparTA Design

Figure 2 summarizes the overall architecture of SparTA. At the core of SparTA is the TeSA abstraction, which augments the existing tensor abstraction with *sparsity attribute* (§3.1). An algorithm designer can specify the sparsity patterns in selected tensors of a deep learning model as "Initial Tensor Sparsity Attribute".

Given the initial sparsity attribute, SparTA performs *attribute propagation* to infer the sparsity attributes of all other tensors in the deep learning model, according to the propagation rules (§3.2). Sparsity attribute propagation exposes more optimization opportunities than the original sparse tensors, as shown, for example, in Figure 1.

After attribute propagation, SparTA runs a multi-pass compilation process to generate efficient end-to-end code (§3.3). Compared to a traditional DNN compiler, SparTA conducts two additional compilation passes to exploit model sparsity fully. The first pass transforms the original execution plan of a DNN model into a new one that takes advantage of the given sparsity patterns. A further compilation pass then performs sparsity-aware code specialization. The awareness of tensor sparsity patterns allow SparTA to generate highly customized code. This process may be iterated for further improvement.

Finally, with the final compiled code, model designers can profile the DNN model to obtain authentic performance metrics, including memory consumption and inference latency. Given the feedback, model designers may further update the sparsity attributes in some tensors and repeat this process iteratively to find the best tradeoff. Thus SparTA enables a feedback loop, facilitating the innovation in model sparsity.

3.1 The TeSA abstraction

TeSA augments a traditional tensor with an additional tensor with the same shape, where each element is a scalar value, representing the sparsity attribute of the corresponding element in the original tensor. This allows a user to specify arbitrary sparsity patterns in a tensor, a key requirement of the evolving research on model sparsity [40, 47, 72]. Figure 3 shows an example of TeSA. The left shows the original dense tensor. The right shows the corresponding sparsity attribute, where one prunes the second row in the tensor, uses 8-bit to quantize the bottom-right element and 4-bit for the remaining elements. This example shows that TeSA can unify tensor quantization and pruning in one abstraction. The unified abstraction facilitates the co-optimization of pruning and quantization, *e.g.,* picking the right block size to cover (represent) the remaining (non-pruned) elements while aligning with low-bit hardware instructions (*e.g.,* `wmma` [5]). With TeSA, SparTA can understand the sparsity pattern at compile time, which enables further optimizations. Note that the sparse attribute will only be used in the compile phase, thus it does not impose additional resource burden to the actual compute phase.

3.2 Sparsity Attribute Propagation

The number of tensors in a deep learning model is usually large. A user can only set the sparsity attribute for a subset of the tensors. To maximize end-to-end sparsity, SparTA performs attribute propagation along the DFG of the DNN model

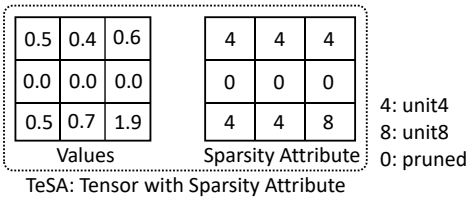


Figure 3: An example of TeSA. Sparsity Attribute denotes the sparsity pattern, including quantization (4 means `uint4`, 8 means `uint8`) and pruning (0 means the element is pruned).

Algorithm 1: TeSA attribute propagation.

```
   Data: G: DFG of TeSA annotated DNN model.
   Result: G with updated TeSAs.
1  Function Propagate(G):
2      S = Set(AllNodesOf(G));
3      while S ≠ ∅ do
4          N = S.PopOne() ;   /* can start from any node */
           /* I_s (O_s) is node N's input (output) TeSA  */
5          I_s, O_s = TeSAOf(N);
6          I_updated, O_updated = PropOneNode(N, I_s, O_s);
7          foreach T ∈ ( I_updated ∪ O_updated ) do
8              B = NeighborNodesOf(T);
9              S.Insert(B.Remove(N));
10     return G;
```

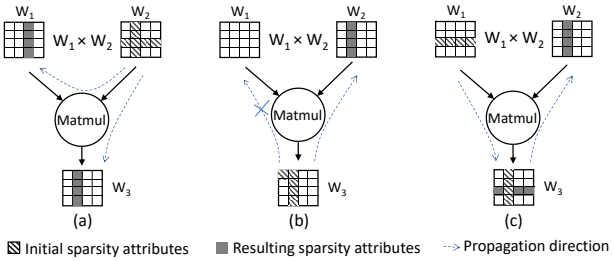


Figure 4: The propagation of sparsity attribute. The gray blocks are propagated sparsity attributes.

to derive the TeSA of other tensors, shown in Algorithm 1. Given a node, if the TeSA of any input or output tensor of a node is updated (`TeSAOf` in Line 5), `PropOneNode` (Line 6) updates the TeSA of other tensors associated with this node, according to a certain propagation rule (as discussed later, the propagation could be bidirectional). The propagation repeats until no TeSA requires further update.

Note that if being propagated multiple times, a sparsity attribute will be updated to increase the sparsity until convergence. Multiple pruning updates lead to the union of the pruned elements in all the updates (The tensor W_3 in Figure 4(c) is an example). For quantization, the attribute will be converged to the fewest quantization bits (or 0-bit, *i.e.,* being pruned). As each propagation monotonically increases sparsity and both the propagations of pruning and quantization are commutative and associative, Algorithm 1 is guaranteed to terminate.

Intra-operator propagation. The propagation behavior of `PropOneNode` varies across different type of operators and attributes. In Figure 4(b), the pruned element [0,0] in tensor W_3 cannot propagate to W_1 and W_2 through the operator `Matmul`, while it does propagate to upstream tensor if the operator is element-wise computation like `ReLU`. Propagation could be bidirectional. Figure 4(a) shows that the input W_2 can affect the output W_3 and another input W_1. And in Figure 4(b), W_2 becomes sparse due to the TeSA of the output W_3.

The sparsity attribute of the quantization type propagates differently. If an output tensor has a low precision (*e.g.,* 4bit) while the input tensor's precision is high (*e.g.,* 16bit), the input may use fewer quantization bits with little impacts on output (*i.e.,* information bottleneck [73]).

Type	Computation	Computation in TeSA Algebra
Unary $f(x) \Rightarrow y$	$sin, cos,$ $\lvert x\rvert, ReLU, \ldots$	$(x=\phi) \rightarrow (y=\phi)$ $(x=\alpha) \rightarrow (y=\alpha)$
Binary $f(x,y) \Rightarrow z$	$+, -$	$((x=\phi) \wedge (y=\phi)) \rightarrow (z=\phi)$ $((x=\alpha) \vee (y=\alpha)) \rightarrow (z=\alpha)$
	$\times, \div, x^y$	$((x=\phi) \vee (y=\phi)) \rightarrow (z=\phi)$ $((x=\alpha) \wedge (y=\alpha)) \rightarrow (z=\alpha)$

Table 2: TeSA algebra on a set of attribute values. ϕ and α represent pruned and non-pruned element respectively.

Next, we show two propagation rules used by SparTA for pruning and quantization attribute. Note that it is possible to extend `PropOneNode` to support more rules as shown in line 27-line 36 of Algorithm 2. New propagation rules can be registered and invoked in `PropOneNode`.

Pruning rule. The propagation of pruning attributes depends on the computation logic of an operator (*e.g.,* $+$, $\times$ in `Matmul`). To capture such property, SparTA defines a TeSA algebra that maps the an operator's element-wise computation to a set with two elements, {*pruned*, *non-pruned*}. The TeSA algebra is shown in Table 2. Given an input TeSA, its output TeSA can be computed using the TeSA algebra, following the same computation flow of the operator. Note that Table 2 can be extended to support new operators.

SparTA also proposes *Tensor Scrambling*, a probabilistic propagation rule that handles black-box or complex operators, where the detailed computation logic is unavailable or unclear. This rule derives the pruned elements of a tensor by scrambling the values of other related tensors. Specifically, the rule sets the pruned elements in the input tensor to zeros, and assigns random values to the remaining elements (*i.e.,* scrambling). It then runs the operator to obtain its output tensor (assuming at least the dense version of the operator is available). By repeating this process enough times (see §5.2), the rule treats those elements that always stay zero as pruned elements in an output tensor.

In addition, the sparsity also propagates from the output to the input, or from one input tensor to another. To achieve this, SparTA leverages the auto differentiation (AD) of DNN computation. An operator's backward operator is also available for the back-propagation in the AD. Let $I_{1\ldots n}$ and $O_{1\ldots n}$ denote an operator's inputs and outputs respectively. Its backward operator's inputs are $I_{1\ldots n}$ and $gO_{1\ldots n}$, with its outputs being $gI_{1\ldots n}$, where the prefix g denotes the gradient of the corresponding tensor. According to AD's property, gI_i and gO_i should have the same TeSA of I_i and O_i (both shape and value). To infer the TeSA propagated from tensor I_i (or O_t) to I_j, SparTA applies TeSA algebra or Tensor Scrambling to the backward operator: using the TeSAs of $I_{1\ldots n}$ and $gO_{1\ldots n}$ as the input, SparTA applies either rule to compute (`PropOneNode`)

Algorithm 2: TeSA attribute propagation rules.

```
   Data: N: a node in DFG, I_s: node N's inputs, O_s: node N's outpus.
   Result: Updated input/ouput TeSAs after propagation on N.
11 Function PruningPropRule(N, I_s, O_s):
12     S_updated = ∅;
13     foreach T ∈ ( I_s ∪ O_s ) do
14         T_updated = TensorScrambling(N, (I_s ∪ O_s)\T);
15         if T_updated != T then
16             S_updated.Update(T_updated);
17     return SplitToIsOs(S_updated);
18 Function QuantizationPropRule(N, I_s, O_s):
19     S_updated = ∅; S = ( I_s ∪ O_s );
20     D_calib = GetCalibrationDataOf(N);
21     foreach T ∈ ( I_s ∪ O_s ) do
22         T_updated = LowerBitAndFinetune(N, S, T, D_calib);
23         if T_updated != T then
24             S.Update(T_updated);
25             S_updated.Update(T_updated);
26     return SplitToIsOs(S_updated);
27 RegisterPropRule(PruningPropRule);
28 RegisterPropRule(QuantizationPropRule);
29 Function PropOneNode(N, I_s, O_s):
30     I_updated = O_updated = ∅;
31     foreach RegisteredPropRule do
32         I_proped, O_proped = RegisteredPropRule(N, I_s, O_s);
33         I_s.Update(I_proped); O_s.Update(O_proped);
34         I_updated.Update(I_proped);
35         O_updated.Update(O_proped);
36     return I_updated, O_updated;
```

gI_j's TeSA, which has the same shape and value of that of I_j.

We use Operator $Y = AX + B$ as an example to illustrate output-to-input and input-to-input propagation, where Y is output tensor, and A, B, and X are input tensors. To derive the TeSA of A, B, X, we take the derivative of the operator with respect to A, B, and X, *i.e.,* $gA = gY \times X^T$, $gB = gY$, $gX = A^T \times gY$, respectively. The sparsity propagation from output tensor Y to input tensor A uses $gA = gY \times X^T$. gY has the same TeSA as Y. Given the TeSA of Y and X, gA's TeSA can be inferred using either TeSA algebra or Tensor Scrambling. The propagation from X to A also uses this backward computation, which is input-to-input propagation. Similarly, the TeSA of B and X can be inferred from Y using $gB = gY$ and $gX = A^T \times gY$, respectively. It is obvious from $gB = gY$ that B has the same TeSA as Y.

The propagation rule of pruned elements can be realized in line 11 of Algorithm 2. Every input/output TeSA of node N is computed (line 14), *i.e.,* propagating the sparsity in other TeSAs (*i.e.,* $(I_s \cup O_s)\backslash T$) to this TeSA (*i.e.,* T). This function returns the updated input and output TeSAs separately (*i.e.,* line 17). Note that here Tensor Scrambling can be replaced with tensor algebra.

Quantization rule. For propagation of quantization attributes, the key is to find tensors with unnecessarily high quantization precision. SparTA defines a quantization rule (line 18 of Algorithm 2) that borrows the idea of knowledge distillation [45,46] to identify such tensors. That is, to identify whether the information passed through an operator can be distilled into a lower precision with acceptable information loss. Since the information loss can be measured through the operator's input and output tensors, we first perform inference on the corresponding DNN model using train/test dataset, and collect the resulting input and output tensors of that operator to construct *calibration data* (line 20). Next, we gradually reduce the quantization precision (*e.g.,* 32-bit to 16-bit) of one tensor of this operator while keeping other tensors unchanged. The operator is then quantized and fine-tuned using the calibration data under the new precision. The fine-tuning is to minimize Mean Squared Error (MSE) between the output tensors in calibration data and the output tensors of that operator after lowering the precision. If the drop of model accuracy is smaller than a predefined threshold (*e.g.,* 1% in our experiment), the new quantization attribute of that operator is accepted (line 22). The process repeats for other tensors in the operator, until all tensors are evaluated. To reduce the cost of collecting calibration data, SparTA works through all the operators in a DNN model in a topological order and caches the activations computed in earlier quantization propagations. Collecting an operator's calibration data only needs partial inference from the nearest cached activations to this operator. For example, consider a sequential model with two layers L_a and L_b. The propagation on L_a has collected its output activations in its calibration data. When working on L_b, its calibration data can be collected by doing partial inference from the collected output activations of L_a. Our evaluation results in §5.2 show the effectiveness of this propagation rule.

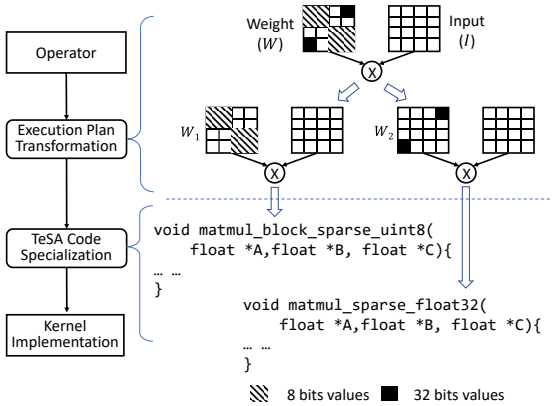


Figure 5: Two-pass compilation to generate an efficient kernel for an operator (`MatMul`).

3.3 Code Generation with TeSA

After attribute propagation, tensors in the DNN model may show a mixture of different sparsity patterns [42, 57, 74, 84]. Such complex patterns make it difficult to generate efficient kernel code. SparTA therefore transforms a tensor with a complex sparsity pattern to multiple tensors, each with a simpler sparsity pattern. Correspondingly, SparTA rewrites the execution plan of the associated operator to accommodate

new operators that compute the transformed tensors. Finally, SparTA performs code generation for the transformed execution plan, with sparsity-aware specialization.

Figure 5 shows an example of such a two-pass compilation process for `Matmul`. The weight tensor W is a tensor with a mixed precision, where two structured blocks use 8bit quantization and one fine-grained element uses 32bit. SparTA transforms W into W_1 and W_2, each using a simpler quantization scheme. Consequently, two operators are introduced to compute $W_1 \times I$ and $W_2 \times I$, respectively, using the hardware instructions fit for the specific sparsity attribute. As a result, the original execution plan with one operator is transformed into a new plan that requires more tensor operations.

Execution-plan transformation. This compilation pass transforms a tensor with a complex sparsity pattern to "regular" (simple) sparsity patterns, which facilitates further optimizations in later passes. In SparTA, a regular sparsity pattern means the TeSA of a tensor shows only one type of quantization attribute and one block size of pruning attribute.

The detailed execution plan transformation for a DNN model is performed operator-by-operator, shown in Algorithm 3. For simplicity, we assume the operator has m inputs and a single output. The process starts from the operator's input and output TeSAs, each of which could be transformed to one or multiple TeSAs using `TransformTeSA`. Correspondingly, the operator is transformed to $|T_o|\prod_{i=1}^{m}|T_i|$ sub-operators, which are the Cartesian product of those decomposed TeSAs. The sub-operator usually has the same computation logic as the original operator (*e.g.,* the operators that can be expressed with *Einstein summation* [7]), an approach that has also been taken in the context of DNN model partitioning [82]. The system performs code generation for each sub-operator (line 45), profiles the resulting kernel in the real hardware, and records the profiled result (line 47).

Note that the transformation is a repetitive search process. Given a TeSA, `TransformTeSA` may have multiple transformation plans. The process iterates over each plan to find the satisfied one (line 38). Figure 6 shows an example. On the left, a mixed precision TeSA can be decomposed to multiple TeSAs each of which has one precision. It can also be transformed to one TeSA where all the elements are aligned to the highest precision. Similarly, for the right example, the sparse TeSA can be decomposed to two TeSAs, one with a block size of 2x2 and the other with a block size of 1x1. It can also be transformed to one TeSA of block size 2x2 with the pruned elements set to zero, or transformed to one TeSA of block size 1x1. Note that the algorithm may decide not to decompose a tensor and choose a block size of 1x1, indicating that the TeSA has a fine-grained sparsity that is hard to be transformed to regular sparsity.

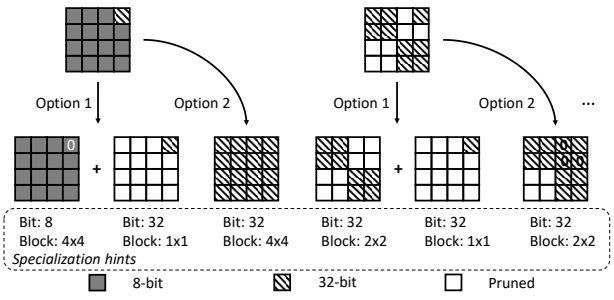


Figure 6: Multiple transformation plans produced by a transformation policy. The specialization hints are used by the second pass compilation for code specialization.

Algorithm 3: Transform an operator's execution plan.

Data: N: An m inputs single output operator to be transformed; P: A transformation policy.

37 **Function** `TransformOp(`N`)`**:**
  `/* HasBudget determines the number of transformation options that can be searched */`
38 **while** *P.HasBudget() and P.PerfNotSatisfied()* **do**
    `/* loop body is one transformation option */`
39 S = [];
40 $I_1, \ldots, I_m$, O = *InputsOutputsTeSAOf*(N);
41 **foreach** $i \in 1, \ldots, m$ **do**
42 $T_i = P$.TransformTeSA(I_i);
43 $T_o = P$.TransformTeSA(O);
44 **foreach** $\langle t_1, \ldots, t_m, t_o \rangle \in T_1 \times \cdots \times T_m \times T_o$ **do**
45 N_{sub} = *SpecializeOp*(op=N, in=$t_1, \ldots, t_m$, out=t_o);
46 S.Append(N_{sub});
47 P.RecordPerf(G=*ComposeToGraph*(S), *Profile*(G));
48 **return** P.BestTransformation();
49 **Function** P`::TransformTeSA(`T`)`**:**
  `/* return a new transformation option per run */`
50 $T_{transformed}$ = [];
  `/* BitOption: (i) 4 and 8, (ii) only 8, if hardware natively supports 4-bit and 8-bit */`
51 $T_1, \ldots, T_k$ = *self*.BitRounding(T, *self*.SampleBitOption());
52 **foreach** $i \in 1, \ldots, k$ **do**
53 $T_i^1, \ldots, T_i^j$ = *self*.WeightedBlockCover(T_i);
54 $T_{transformed}$.Extend($T_i^1, \ldots, T_i^j$);
55 **return** $T_{transformed}$;
56 **Function** P`::WeightedBlockCover(`T`)`**:**
57 B_{chosen} = [];
  `/* covering non-pruned elements to blocks using every available block size, to produce` B `*/`
58 B = *self*.AllCoveredBlocks(T, *self*.AvailableBlockSizes());
59 **while** *not AllElementsCovered*(T, B_{chosen}) **do**
60 B_{cost} = *self*.UpdateBlockCost(B);
61 b = *BlockWithMinCost*(B_{cost});
62 B_{chosen}.Append(b);
63 $B = B - b$;
  `/* decompose` T `to the TeSAs with different block sizes based on` B_{chosen} `*/`
64 **return** *DecomposedTeSAs*(T, B_{chosen});

`TransformTeSA` (line 49) implements the logic of transforming a TeSA. It first decomposes TeSA in `BitRounding`, based on both the quantization bit width that the TeSA contains and the possible quantization bit width supported by the hardware. For example, if the hardware supports both 4-bit and 8-bit instructions, there are at least two rounding options: (i) rounding to 4-bit and 8-bit accordingly, (ii) all rounding

to 8-bit. For each TeSA returned by `BitRounding`, function `WeightedBlockCover` chooses one or multiple proper block sizes to cover the non-pruned elements, which we treat as a *weighted set cover* problem [19]. The weight of each block size corresponds to the cost of computation with the block size on the underlying hardware (see §4 for details). We use a simple greedy algorithm to pick the blocks with the lowest cost (*i.e.,* $\frac{block\_weight}{num\_covered\_elements}$), until all non-pruned elements are covered (line 56).

Transformation policy P can be further customized to incorporate new optimizations (*e.g.,* supporting Sparse Tensor Cores [1] detailed in §5.3).

To help codegen, each transformed TeSA is attached with the information about the bit width and block size, named as *specialization hints* (illustrated in Figure 6). The hints will be passed to the second pass, elaborated next.

TeSA code specialization. The second compilation pass specializes kernel code for each (sub)operator (*i.e.,* line 45 in Algorithm 3). The specialization hints generated from the previous pass guide the specialization strategy. For example, the bit-width of an operator suggests whether to leverage a specific hardware instruction (*e.g.,* `DP4A`). And the loop tiling of the operator should be aligned with the block size for effective dead code elimination (DCE). In addition, SparTA can leverage traditional DNN compilers for dense computation. For example, some intra-block computation is dense and thus can use a dense implementation generated by existing DNN compilers [29, 60, 91].

The specialization process starts from a dense version of the operator, implemented as multi-level loops generated by a traditional DNN compiler [29]. It first specializes under the guidance of the block size in the specialization hints. It searches from the outermost loop until the level (say l) of inner loop body aligns with the block size. Since the pruning sparsity attribute is specified at the granularity of block size, many runs of the loop body within level l are dead computation. To eliminate the dead computation, we introduce a new schedule primitive `dismantle` that jointly performs loop unrolling and DCE. When `dismantle` is applied on a loop, this loop and all its outer loops are unrolled and specialized with the given sparsity attribute. An example is shown in Figure 7(b). `dismantle` is applied on the third loop, thus the top three loops are unrolled, generating eight small Matmuls (*i.e.,* [2,2]x[2,2]). According to the sparsity pattern in Figure 7(a), six of them are dead computation and can be eliminated. In essence, `dismantle` embeds a specific sparsity pattern into the code, which eliminates the need of sparsity encoding, *e.g.,* compressed sparse row (CSR) [26]. As the index to the non-pruned blocks/elements is specialized in the code, the overhead of indirect addressing on the index is removed.

Given a different transformation plan (and the specification hints), the code can be specialized differently. The hint in Figure 7(c) show a smaller block size. In this case, the loop body is a smaller Matmul (*i.e.,* [2,1]x[1,1]), which enables

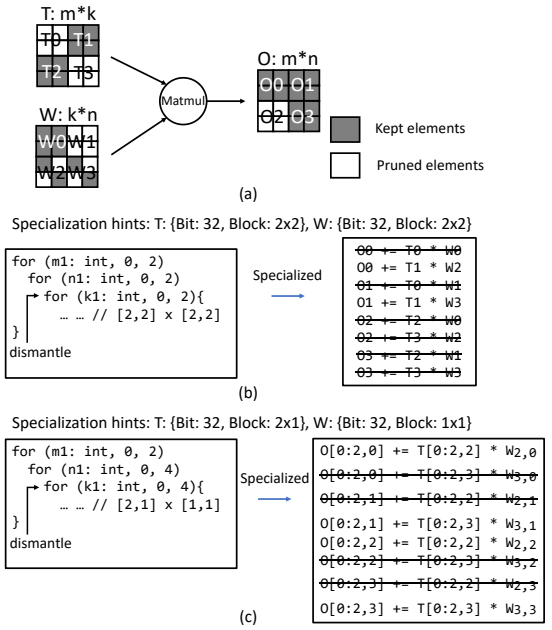


Figure 7: Sparsity-aware code specialization, leveraging specialization hints generated during execution plan transformation. (a) is a sparse Matmul, (b) and (c) are its specialized kernel code with different transformation plans. $T[x:y,z]$ denotes the elements on row x to y and column z, $W_{x,y}$ denotes the value on row x and column y of W.

more DCEs. Besides the dead computations eliminated in the previous case, four computations of the small Matmul can be removed. Furthermore, as the small Matmul only accesses one value in W, the value can be directly specialized to the code (*i.e.,* $W_{x,y}$) without maintaining a sparse tensor W in memory.

A specialization hint could also specify the block size equal to the tensor's shape (*i.e.,* one block covers the whole tensor). In this case, SparTA can directly use the existing state-of-the-art general sparse kernel implementation (*e.g.,* cuSPARSE [3], taco [53]) or even use the dense kernel implementation if it perform better. SparTA's specialization framework is general to incorporate any sparsity-aware techniques, including the off-the-shelf sparse kernel and even its dense version.

Specializing operators with quantization attributes also works on the multi-level loops, but starting from the innermost loop. SparTA picks a proper hardware instruction based on the bit-width denoted in TeSA, *e.g.,* `DP4A` [13] or `wmma` [5] for 8bit, `wmma` for 4bit. The specialized tiling of the innermost loop(s) is then aligned to the computation shape of the instruction. For example, one supported computation shape of `wmma` is the Matmul [16,16]x[16,16]. To specialize for this instruction, the tiling should rearrange the innermost loop body to align with the shape, and then replace the rearranged loop body with the instruction. The tiling for the instruction `DP4A` with a shape [1,4]x[4,1] can be done similarly.

4 Implementation

We implemented SparTA on Rammer [60], a state-of-the-art open-source DNN compiler with no special support for sparsity. We implemented sparsity attribute propagation as a dedicated compilation pass over Rammer. A DNN model is converted to an ONNX graph [14] before compiling. Each TeSA element has a two-byte attribute: 7 bits record the bit width of the element, 4 bits specify the element's data format (*e.g., unsigned int*, *float32*, *bfloat16*), and the rest 5 bits are reserved. The bit-width zero means the element is pruned. TeSA exists only during compile time and therefore incurs no runtime memory cost. We implemented the execution plan transformation within Rammer, with an additional compilation pass that rewrites the graph with a better execution plan. The specialized sub-operators are injected into Rammer's kernel DB for constructing the whole executable.

For the efficient execution of weighted block cover in execution plan transformation (§3.3), SparTA calculates the weight of different block sizes. Specifically, we implemented a kernel template for block sparsity, and evaluated 13 different block sizes on that template. The overhead of each block size is profiled by measuring the latency of the kernel with zero sparsity and dividing the latency with the number of blocks. The overhead is used as the weight of the block sizes. When an operator is too sparse to saturate available cores, the weights may become less accurate. In such cases, we enumerate all the combinations of block sizes to run the weighted block cover algorithm and pick the best one. For kernel specialization, we implement the *dismantle* primitive based on loop unrolling. When a loop is unrolled with *dismantle*, we read the corresponding TeSA and eliminate dead computations accordingly.

SparTA, as a full-stack solution for model sparsity, has supported 21 model sparsity algorithms, including 16 pruning algorithms and 5 quantization algorithms (full list omitted due to page limit). Those algorithms can run on SparTA with little code modifications, and benefit from SparTA not only on sparsity exploration but also on model fine tuning, which will be demonstrated in §5.4.

5 Evaluation

We evaluate SparTA on three popular DNN models with four different sparsity patterns on NVIDIA GPU, AMD GPU, and Intel CPU. Overall, our key findings include:

- SparTA significantly reduces the inference latency of sparse DNN models with less memory consumption. The speedup is up to 10.6x, 5.0x, 7.5x, 20.1x, 5.8x, 5.6x, 1.7x over PyTorchJIT, TensorRT, TVM, TVM sparse[2], Rammer, Rammer sparse[3], and OpenVINO (CPU), respectively. The average speedup is 3.8x, 2.6x, 4.2x, 8.4x, 3.0x, 3.2x, 1.7x. (§5.1)

[2]Excluding cases where kernel tuning failed for TVM and TVM sparse.
[3]The state-of-the-art sparse kernels wrapped in Rammer.

Model	Type	Sparsity	Ratio	Acc (%)
BERT [34]	NLP {*Matmul*}	Structured [87]	95%	89.7->88.49
		Unstructured [43]	95%	89.7->88.67
		Structured+8bit [52]	95%	89.7->88.03
		Mixed Sparsity [48, 84]	94.99%	89.7->88.63
MobileNet [49]	CV {*Conv*}	Structured [94]	60%	78.27->75.62
		Unstructured [43]	95%	78.27->64.15
		Structured+8bit [52]	60%	78.27->75.13
HuBERT [50]	Speech {*Conv*, *Matmul*}	Structured [56, 62]	80%	95.61->95.1
		Unstructured [43]	95%	95.61->95.55
		Structured+8bit [52]	80%	95.61->94.3

Table 3: Evaluated DNN models with different sparsity patterns and their resulting accuracy. The second column lists the major operators of each model. The column **Ratio** denotes the initial sparsity ratio for pruned weights.

- Sparsity attribute propagation increases the end-to-end sparsity ratio by up to 39.7%. With execution plan transformation and code specialization, SparTA can achieve up to 6.7x speed up over the state-of-the-art sparse kernel implementation for a sparse DNN operator (*e.g.,* Matmul) with complex sparsity patterns. (§5.2, §5.3)
- SparTA facilitates the development and exploration of sparse DNN models, producing DNN models with lower inference latency and/or higher accuracy. (§5.4)

5.1 End-to-End Experiments

We evaluate SparTA on the inference latency and memory usage of three popular DNN models across different task domains, shown in Table 3. We evaluate four representative sparsity patterns, covering different pruning and quantization schemes and their combination. *Unstructured* sparsity prunes model weights in the granularity of an element in weight tensors to reach the desired sparsity ratio [43, 54, 55]. *Structured* sparsity prunes weights in the granularity of column, row, channel, or block, depending on specific models [44, 56, 59]. We apply different sparsity patterns to the three selected models to show SparTA's effectiveness under various patterns. BERT is pruned in row combined with a block of size 32x32 [87]; MobileNet gets pruned in the output channel [94]; for HuBERT, it is a combination of channel pruning in the Conv layer and head pruning in the transformer layer [56, 62]. To further demonstrate the powerful expressiveness of TeSA, we apply structured sparsity, based on which 8bit quantization is further applied on the remaining tensor elements to construct the third sparsity pattern, *i.e., Structured+8bit*. Finally, we introduce an even more complicated *Mixed Sparsity* for BERT. On top of the *Structured+8bit* sparsity, we apply unstructured sparsity with 32bit quantization back to 0.01% of the pruned elements [48, 84]. This leads to a total sparsity ratio of 94.99%.

We trained the models (BERT on dataset QQP [51], MobileNet on ImageNet-Dogs [33], HuBERT on SUPERB [85]) applied with the above sparsity patterns, the accuracy change

of those sparse models is shown in the **Acc** column of Table 3. Overall, the accuracy drops are consistent to those reported in the corresponding papers. The accuracy of sparse BERT drops around 1%. Mixed sparsity has the accuracy of 88.63% at 94.99% sparsity, nearly the same accuracy as unstructured sparsity, but is much easier to be accelerated. For MobileNet, unstructured sparsity has a higher accuracy drop, as its sparsity is high (*i.e.,* 95%). For HuBERT, 95% unstructured sparsity can outperform the structured ones with 80% sparsity ratio.

The models are evaluated on three types of accelerators: NVIDIA GeForce RTX 2080 Ti, AMD Radeon VII, and Intel Xeon Silver 4210 CPU. We compare SparTA with seven representative solutions, including one popular deep learning framework: PyTorch (v1.7) with JIT [67], two vendor-specific toolkits: TensorRT (v7.2) for NVIDIA GPUs [18] and OpenVINO (v2021.4.1) for Intel CPUs [15], two DNN compilers: TVM (0.9.dev0) [29] and Rammer [60] (which offers the state-of-the-art performance). To evaluate the state-of-the-art sparse kernels/libraries in an end-to-end model, we create Rammer sparse (or Rammer-S) by wrapping in Rammer these sparse kernel libraries/implementations, including cuSPARSE [3], taco [53], and Sputnik [39] for NVIDIA GPU, hipSPARSE [17] for AMD GPU, MKL Sparse Linear Algebra [12] for Intel CPU. For TVM, we also evaluate its sparsity support [22] (denoted by TVM-S). Each model on TVM is tuned with 1,000 trials per task using Ansor [91], aligned with the common practice [91]. The batch size we used in the end-to-end experiments (except Figure 12) is 32.

5.1.1 SparTA on CUDA GPUs

Structured sparsity. The first row of Figure 8 shows the inference latency of the three models on the structured sparsity. PyTorch, TensorRT, TVM, and Rammer treat them as three dense models. TensorRT performs the best among them. Compared to TensorRT, SparTA is 3.7x, 2.9x, 2.4x faster on BERT, MobileNet, and HuBERT, respectively. TVM-S and Rammer-S are aware of sparsity. TVM-S incurs high inference latency, as the kernel templates it uses cannot efficiently support different sparsity patterns. Rammer-S performs marginally better than TensorRT on MobileNet and HuBERT. The SOTA sparse kernel uses Sputnik, which performs better than cuSPARSE and taco on those models. SparTA performs 1.7x, 2.6x, and 2.3x faster than Rammer-S. Its performance gain comes mainly from sparsity propagation, which increases the whole model's sparsity (see §5.2) and sparsity transformation, *i.e.,* covered with different block sizes on different layers (see §5.3).

Memory footprints in the inference are shown in the first row of Figure 9. SparTA shows the smallest footprint. For MobileNet, PyTorch and TensorRT consume much more memory, because they use cuDNN, which requires additional memory to store weights and activations. SparTA's memory usage is smaller than TVM-S and Rammer-S due to sparsity propagation, which increases the sparsity ratio.

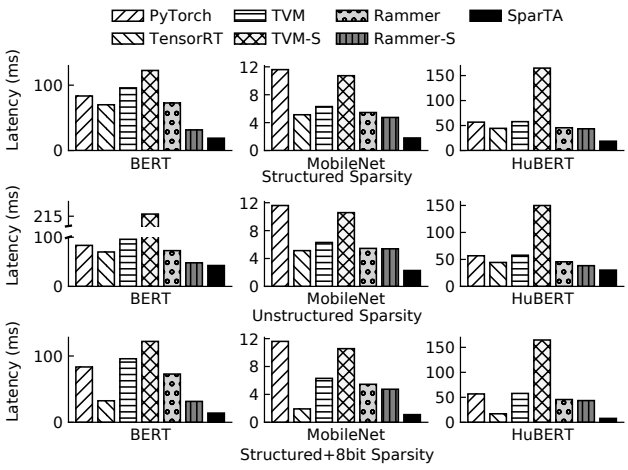


Figure 8: Inference latency of different models with three sparsity patterns on NVIDIA 2080 Ti.

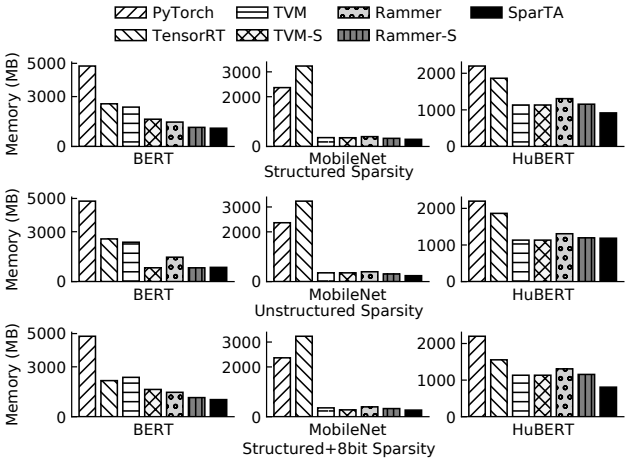


Figure 9: GPU memory usage of different models with three sparsity patterns on NVIDIA 2080 Ti.

Unstructured sparsity. For unstructured sparsity (*i.e.,* second row of Figure 8), TensorRT also performs the best among those dense baselines, marginally better than Rammer. SparTA is 1.6x, 2.2x, 1.5x faster than TensorRT on BERT, MobileNet, HuBERT, respectively. Rammer-S still uses Sputnik. SparTA outperforms Rammer-S by 1.13x, 2.4x, 1.3x on BERT, MobileNet, HuBERT, respectively. The speedup on MobileNet is high because the sparsity is easier to be propagated on depthwise and pointwise convolution even with unstructured sparsity. On BERT and HuBERT, the performance gain over Rammer-S mainly comes from code specialization (*i.e.,* weight values are embedded into kernel code). For the memory usage (*i.e.,* the second row of Figure 9), SparTA shows a usage similar to TVM-S and Rammer-S, and performs better than the other baselines.

Structured+8bit. Shown in the third row of Figure 8, Ten-

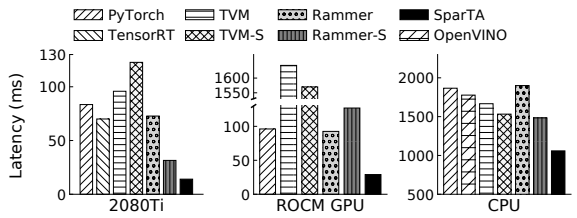


Figure 10: Mixed Sparsity end-to-end latencies.

sorRT performs much better than PyTorch, TVM, and Rammer, because it supports low-bit computations with Tensor Cores. SparTA outperforms TensorRT by 2.3x, 1.7x, 2.2x on BERT, MobileNet, HuBERT, respectively. This is because, besides leveraging the hardware instruction `wmma` of Tensor Core, SparTA further combines the optimization for structured sparsity. Compared to Rammer-S (also using Sputnik), SparTA is 2.3x, 4.3x, 5.6x faster on BERT, MobileNet, HuBERT, respectively, because Rammer-S has limited optimization (*e.g.,* missing low-bit instructions) for each single sparse operator, while SparTA does holistic optimizations on the model, *e.g.,* sparsity propagation, operator transformation and specialization. The speedup shows the combined gain from structured sparsity (*i.e.,* the first row of Figure 8) and low-bit instructions. The memory usage of SparTA (*i.e.,* the third row of Figure 9) is the lowest: the memory saving comes from both low-bit values and pruned elements. This highlights the benefit of SparTA and in particular TeSA that uses a unified abstraction for pruning and quantization to make such a joint optimization possible.

Mixed sparsity. The left figure in Figure 10 shows the latency of BERT with Mixed Sparsity. SparTA is 5.9x, 5.0x, 6.8x, 8.7x, 5.2x, 2.2x faster than PyTorch, TensorRT, TVM, TVM-S, Rammer, Rammer-S, respectively. Unlike structured+8bit, TensorRT shows slight advantage over other baselines on mixed sparsity, although most elements are 8-bit.

Latency Breakdown. Figure 11 shows the performance breakdown of BERT on the four sparsity patterns. "+Sparse Kernel" applies our generated sparse kernels following the original sparsity ratio without operator transformation and kernel specialization. It can be treated as Rammer-S. "+Propagation" applies sparsity propagation on the model and regenerates the sparse kernels without transformation and specialization. "+Transformation" tunes the block size for covering non-pruned elements of each sparse operator, and for mixed sparsity it also decomposes sparse tensors to multiple ones. "+Specialization" tunes intra-block implementation and embeds values into codes when necessary.

For mixed sparsity, the latency reduction brought by each optimization is 55.8%, 19.7%, 37.7%, and 12.6%, respectively. The other three sparsity patterns could be viewed as a type of breakdown of mixed sparsity. In structured sparsity and structured+8bit, transformation brings 20.5% and 26.5% latency reduction, respectively, while propagation brings 19.7% and 15.8% latency reduction, respectively. Finally, intra-block specialization brings 8.2%, 11.4%, and 13.7% latency reduction for structured, unstructured, and structured+8bit, respectively. The significance of a certain optimization depends on DNN models and sparsity patterns. For BERT in Figure 11, "+Sparse Kernel" brings 2.1x gain on average, SparTA brings an extra 2x gain. For MobileNet to be illustrated in §5.2, "+Propagation" brings the most gain (*e.g.,* increasing sparsity from the 50% to 89.7%).

Latency of different batch sizes. Figure 12 shows the performance of BERT under different batch sizes on NVIDIA 2080Ti. When batch size varies from 8 to 64, SparTA is on average 4.1x-4.6x, 2.4x-2.7x, 4.2x-6.3x, 5.9x-13.8x, 3.6x-4.1x, 2.2x-2.6x faster than PyTorch, TensorRT, TVM, TVM-S, Rammer, Rammer-S, respectively on three sparsity patterns. The overall speedup of SparTA is similar across different batch sizes. The range of the speedup over TVM-S is relatively large, because large batch size induces large tuning space that makes the kernel tuning in TVM less effective.

Compiling overhead. The overhead of SparTA comes from the compiling phase, which consists of three parts: propagation, transformation, specialization. The overhead is positively related to the number of operators in the model. Taking BERT as an example, the propagation, transformation, and specialization take 3 minutes, 2 hours and 1.5 hours respectively using a single thread. It is possible to reduce the overhead by leveraging more prior knowledge in transformation policy and pre-tuned kernels. We leave it as future work.

5.1.2 SparTA on Other Accelerators

ROCm GPU. Figure 13 shows inference latency of the three models on AMD Radeon VII. The speedup of SparTA over PyTorch on the three sparsity patterns is up to 3.5x, 4.2x, 2.2x for BERT, MobileNet, and HuBERT, respectively. The kernel tuning of TVM on ROCm GPUs does not function properly (always stuck in Debug mode), the performance of TVM and TVM-S on BERT and HuBERT suffers a lot. They show reasonable performance on MobileNet, because they

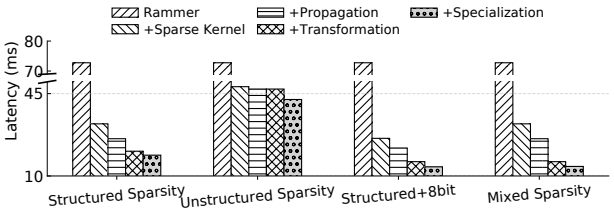


Figure 11: Performance breakdown of SparTA for different sparsity patterns of BERT on 2080 Ti. Each bar shows the result of applying the additional optimization labeled on this bar from the previous one.

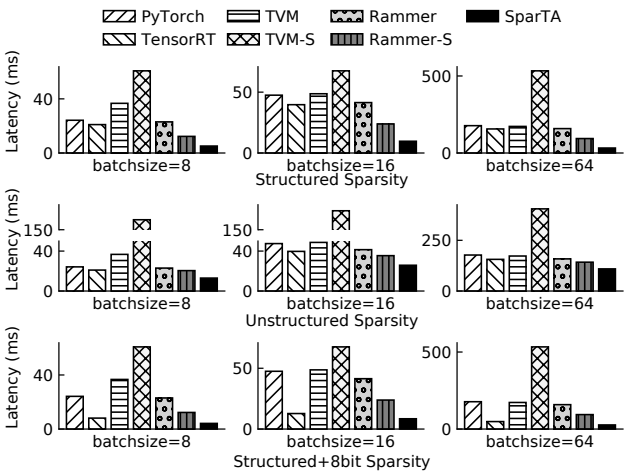


Figure 12: Inference latency of BERT under different batch sizes on NVIDIA 2080Ti.

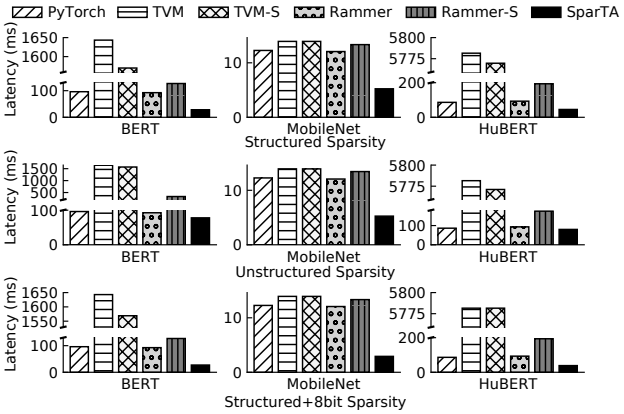


Figure 13: Inference latency of different models with three sparsity patterns on AMD Radeon VII.

have provided reasonably good default kernel schedules for some popular DNN models including MobileNet. Compared to Rammer, SparTA is up to 3.4x, 4.1x, 2.4x faster for BERT, MobileNet, and HuBERT, respectively, on the three sparsity patterns. Rammer-S uses hipSPARSE on ROCm GPU, the speedup of SparTA over Rammer-S is up to 4.7x, 4.6x, 5.0x for BERT, MobileNet, and HuBERT, respectively.

For mixed sparsity of BERT shown in the middle of Figure 10, SparTA is 3.3x, 56.7x, 54.1x, 3.2x, 4.4x faster than PyTorch, TVM, TVM-S, Rammer, and Rammer-S, respectively. Although Rammer-S with hipSPARSE has higher latency than Rammer, it has a lower memory footprint.

Intel CPU. We evaluated mixed sparsity pattern of BERT on CPU, the result is shown in the right of Figure 10. Compared to OpenVINO, a high-performance inference engine for Intel CPUs, SparTA achieves 1.7x speedup. For PyTorch, TVM, TVM-S, the speedup of SparTA is 1.8x, 1.6x, 1.5x, respectively. Rammer-S uses the MKL library, which leverages

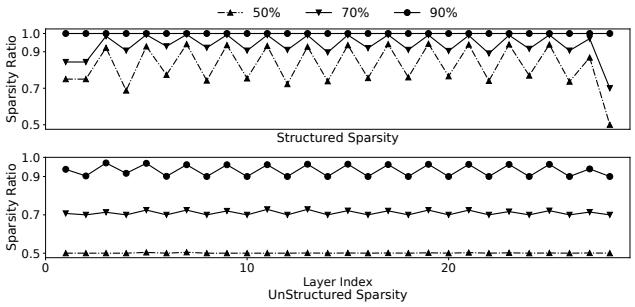


Figure 14: Propagated sparsity across the layers for different sparsity patterns on MobileNet

the sparsity, thus faster than other baselines. SparTA still has 1.4x performance gain over Rammer-S, because it leverages low-bit instruction (*i.e.,* AVX512 VNNI [10]) and further optimizes the model with sparsity propagation and execution plan transformation in a holistic way.

5.2 Sparsity Attribute Propagation

Propagation of pruned elements. The performance gain brought by propagation on BERT has been illustrated in Figure 11. The propagation has higher potentials on MobileNet, as convolution's filter size is small (*e.g.,* 3x3). Figure 14 shows how sparsity is propagated across layers on MobileNet, which increases each layer's sparsity ratio. In this experiment, we tested three sparsity ratios (*i.e.,* 50%, 70%, 90%) pruned by the same algorithm used in the end-to-end experiment. For each sparsity ratio, we prune every layer of MobileNet to the target ratio. Then the propagation rule is applied. The accuracy results of inference on train/test dataset are exactly the same before and after propagation, as the propagation rule for pruned elements does not affect computation logic.

For structured sparsity, the total sparsity ratio is increased from 50% to 89.7% after propagation. The curve's zigzag is caused by different propagation potential of the interleaving depthwise convolution and pointwise convolution in MobileNet. Interestingly, when the original sparsity ratio is 90%, after propagation the sparsity ratio becomes 100%, which explains the anomaly that, although there are 10% filters left on each convolution (before propagation), the model's accuracy is similar to a random image classifier. The propagation ability on unstructured sparsity is lower. Only high sparsity ratio could bring an obvious increase of sparsity ratio. For example, with 90% original sparsity, the total sparsity is increased to 95.3% after propagation. With Tensor Scrambling, our experiences show 256 randomly sampled tensors can identify sparsity correctly.

Propagation of quantization bit. In this experiment, we evaluate the propagation rule for quantization described in §3.2. We follow the same approach proposed in HAQ [77] to quantize MobileNet. Specifically, it uses reinforcement learn-

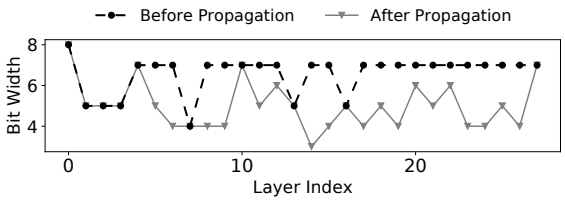


Figure 15: The quantization (bit width) of each layer in MobileNet before and after propagation.

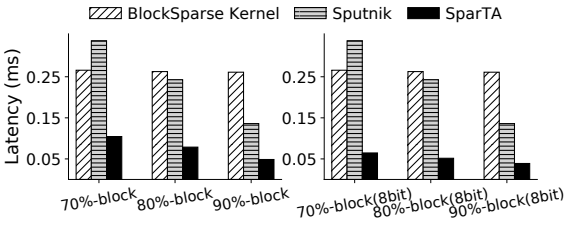


Figure 16: The performance of execution plan transformation for mixed sparsity patterns. B is sparsified for the matrix multiplication $A \times B$ (1024x1024x1024). "X%-block" means X% block sparsity mixed with 1% unstructured sparsity.

ing to explore the bit width on each weight and activation tensor. The candidate bit width is between 0 to 8. After exploring 50 different configurations of bit width, we pick the best one, whose accuracy on ImageNet is 64.6%. The propagation rule is then applied to that configuration. The experiment result is shown in Figure 15: 18 out of 28 layers reduces its bit width (from 7bit to around 4bit), while the model accuracy after propagation only drops slightly, from 64.6% to 64.2%. From another point of view, our propagation rule for quantization is complementary to the search algorithm (*e.g.,* reinforcement learning, simulated annealing [58]) on quantization bits. A proper combination of them could improve search efficiency, which is an interesting future work.

5.3 Efficient Code Generation with TeSA

Effectiveness of execution plan transformation. The sparsity-aware execution plan transformation in SparTA could handle complex sparsity patterns efficiently. We test two sophisticated sparsity patterns: (1) Mix of structured sparsity with block size 32x32 and unstructured sparsity (*i.e.,* 1x1) [48, 53]. There are 1% unstructured elements, and the structured sparsity ratio varies from 70% to 90%. (2) Based on the first sparsity pattern, we further make the structured sparsity 8bit, and make unstructured sparsity 32bit [84]. To show the effectiveness of transformation, we compare SparTA with two baselines: one is our specialized kernel for structured sparsity (denoted by BlockSparse), where the unstructured elements are covered with 32x32 blocks; the other is Sputnik, which is optimized for unstructured sparsity.

The results are shown in Figure 16. For the first sparsity pattern, SparTA transforms the operator into two sub-operators with structured sparsity and unstructured sparsity, respectively. After transformation, SparTA becomes 2.5x, 3.3x, 5.4x faster than BlockSparse on the three sparsity ratios, respectively. The performance of BlockSparse has little change, because the blocks to cover those unstructured elements construct the major of the blocks in the computation. The speedup of SparTA over Sputnik is 3.2x, 3.1x, 2.8x, respectively. Sputnik performs the worst on 70%-block, because it treats each block as 1,024 unstructured elements, missing optimization opportunities. For the second sparsity pattern, the performance gain of SparTA is much higher, *i.e.,* 4.1x, 5.1x, 6.7x faster than BlockSparse, 5.2x, 4.7x, 3.5x faster than Sputnik. This is because SparTA further leverages low-bit instructions for the computation of those 32x32 blocks.

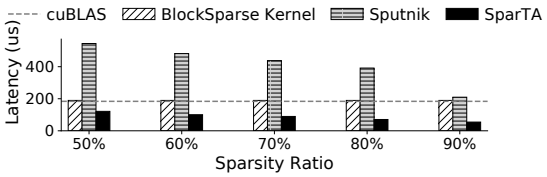


Figure 17: The performance of execution plan transformation leveraging Sparse Tensor Cores. B is sparsified with different sparsity ratios for $A \times B$ (1024x1024x1024).

The transformation can effectively leverage special hardware like Sparse Tensor Core [1]. Sparse Tensor Core has a strict requirement on tensor's sparsity pattern, *e.g.,* one element should be pruned in a $[1 \times 2]$ tile (50% sparsity ratio). To leverage Sparse Tensor Cores for the generic unstructured sparsity, we develop a new transformation policy to decompose an unstructured sparse tensor into two: one follows the sparsity requirement of Sparse Tensor Cores, the other contains the elements not included in the first one. The first one uses Sparse Tensor Cores, while the other uses our specialized sparse kernel. To evaluate the transformation, we randomly generate an unstructured sparse tensor whose sparsity ratio ranges from 50% to 90% in one input of Matmul. The experiment runs on NVIDIA A100, the result is shown in Figure 17. SparTA performs better than both BlockSparse and Sputnik, as it leverages cuSPARSELt [6], a library optimized for Sparse Tensor Cores, for sub-Matmul that costs around 40 us. The other sub-Matmul has 12.5%, 8%, 4.5%, 2.0%, 0.5% sparsity ratio respectively. BlockSparse shows a similar performance to cuBLAS. As the sparsity is randomly introduced, it actually computes a dense Matmul.

Sparsity Pattern	1	2	3	4	5
Origin Latency(ms)	1.293	0.361	2.808	1.263	0.189
SparTA Latency(ms)	0.436	0.191	0.599	0.569	0.101
Best Block Size	32x128	128x32	32x128	32x64	128x64

Table 4: Block size transformation

During the transformation, SparTA also finds the best

block size to cover those non-pruned elements. We picked 5 sparse tensors with different sparsity patterns in BERT, apply `WeightedBlockCover` to find the best block size of the 5 tensors. Table 4 shows the found block sizes. The chosen block sizes are all different from the original 32x32 block size and they all perform much better than the kernel implemented with the original block size. Essentially, the block covering makes a trade-off between the efficiency that a certain block size is optimized for the underlying hardware and the ratio of the computation wasted using that block size.

Effectiveness of TeSA code specialization. We evaluate SparTA's specialized matrix multiplication kernel under different unstructured sparsity ratios, ranging from 50% to 99%. We compare the specialized kernels with cuSPARSE, taco [16, 53], and Sputnik [39]. The result is shown in Figure 18. At 99% sparsity, cuSPARSE outperforms cuBLAS, but incurs 2.2x slowdown at 95% sparsity. In most cases, cuSPARSE performs much worse than cuBLAS on latency, although it has a lower memory footprint due to encoded sparse tensors. taco performs worse than cuSPARSE due to its inefficient utilization of shared memory [70]. It is 15.6x slower than cuSPARSE for 99% sparsity; the slowdown is reduced to 4.0x when the sparsity is 50%. SparTA is up to 6.0x faster than cuSPARSE. It outperforms cuBLAS when the sparsity is only 70%. Sputnik also performs better than cuSPARSE and taco. SparTA is up to 1.7x faster than Sputnik.

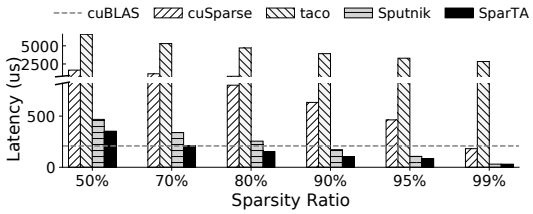


Figure 18: Comparison of cuSPARSE, taco, Sputnik, and SparTA on matrix multiplication (1024x1024x1024) with fine-grained sparsity under different sparsity ratios. B is sparse for $A \times B$.

5.4 Augmented Model Sparsity Exploration

SparTA, as a full-stack solution for model sparsity, facilitates the exploration of existing model sparsity algorithms. In this section, we demonstrate this from the following two aspects.

Actual latency vs. FLOPS as proxy-metric for latency reduction in model pruning. In this experiment, we use Simulated Annealing [58] to prune MobileNet to reduce 30% and 40% inference latency, respectively, *i.e.,* the two dash lines in Figure 19. Our baseline uses FLOPS as the metric to filter out the disqualified models: the model whose FLOPS is larger than 70% of the original FLOPS. In contrast, SparTA uses the real latency to filter models. The result is shown in Figure 19. The best sparse models found by the two approaches have similar accuracy. However, the model found via FLOPS does not meet the latency target, 23.8% and 51.4% higher than the target, respectively. This shows FLOPS cannot faithfully reflect real inference latency. In contrast, the sparse models found by the algorithm on SparTA successfully satisfy the latency requirement.

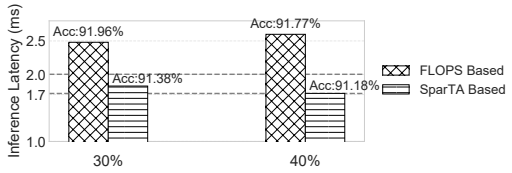


Figure 19: The comparison of using real latency or FLOPS as metric to explore sparse models by Simulated Annealing.

Speeding up sparsity exploration. With high-performance sparse kernels, SparTA can speed up the exploration process of a sparsity algorithm, which usually searches for a sparsity pattern iteratively [58, 86]. In each iteration, the algorithm "sparsifies" a proportion of the model (*e.g.,* 30%) and fine-tunes it. It repeats the iteration until achieving the targeted sparsity (*e.g.,* 90%). In this process, model fine-tuning consumes significant exploration time. With SparTA, model fine-tuning can be accelerated. Figure 20 runs Simulated Annealing, an iterative sparsity algorithm, on ResNet50. The algorithm prunes 50% of the remaining weights and fine-tune 300 epochs in each iteration. SparTA reduces 31.8% of the total exploration time, compared to the baseline that always uses the original dense model.

5.5 Accelerating Sparse Model Training

In addition to model pruning and quantization, some DNN models are designed to be sparse from the beginning, *e.g.,* sparse attention [72]. SparTA can also be used to speed up the training process of such sparse models.

We show this by applying SparTA to the training of NÜWA [81], a state-of-the-art visual synthesis pretrain model that adopts a novel 3D Nearby Attention (3DNA) mechanism. In 3DNA, each token computes the attention to the nearby tokens within a small 3D window, instead of to all the tokens (*i.e.,* full attention).

We implement 3DNA using SparTA and compare the performance with its previous PyTorch implementation (a dense version), and another version implemented using OpenAI's Triton (v1.1.1) [21], a compiler that supports sparse attention. As the two baselines are PyTorch-based, we integrate SparTA-based 3DNA into PyTorch for a fair comparison. The result is shown in Figure 21. Both Triton and SparTA perform much faster than the default PyTorch version, and consume less GPU memory. The default PyTorch version encounters out-of-memory when the batch size grows beyond 16. SparTA is 2.15∼2.24x faster than Triton across different

Figure 20: The improvement on exploration time when using SparTA-accelerated sparse model.

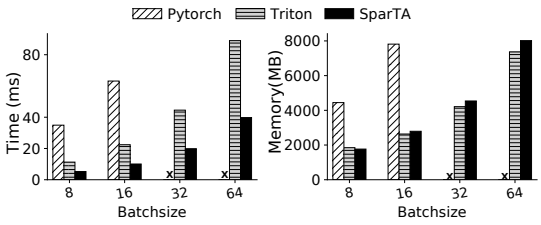


Figure 21: The time of training one batch and GPU memory consumption of 3DNA on NVIDIA 2080Ti.

batch sizes, mainly because SparTA specializes the block sizes (*e.g.,* 32x64, 32x32) covered on non-zero values in the matrix multiplications of 3DNA. The memory usage of SparTA and Triton are similar. As the SparTA-based version currently relies on PyTorch for the management of some intermediate tensor, it is possible to further improve the memory usage by moving it to the SparTA side.

6 Related Works

Sparsity support in DNN frameworks and compilers. Deep learning frameworks like PyTorch [67] and TensorFlow [23] or compilers like TVM/Ansor [29, 91] exploit sparsity by vendor-specific libraries like cuSPARSE/cuSPARSELt [3] or user-provided sparsity kernel templates [29]. The lack of understanding to the specific sparsity pattern across a sparse model leads to a subpar performance. In contrast, with TeSA, SparTA can capture arbitrary sparsity patterns and enable various sparsity-aware optimizations to generate efficient end-to-end code.

SparTA's design incorporates several classic compiler techniques. For example, sparsity attribute propagation is similar to type qualifiers [38] and type inference [32]. OpenMP [35] also leverages attribute propagation in a different problem domain with a different mechanism. Code specialization based on value profiling [27] is also a well-known technique. Zeroploit [69] and PGZ [71] also use a similar idea, but focus on gaming applications. Instead of values, SparTA uses more general attributes for code specialization. And SparTA offers a complete framework for DNN model sparsity.

Sparsity acceleration of DNN models. Sparse matrix multiplication has been studied for decades in scientific computing [68,80]. With the emerging accelerators (*e.g.,* GPU [8,20], TPU [4], FPGA [11], GraphCore [9]), some research optimizes sparse matrix multiplication for a certain type of hardware [24, 26, 39, 80, 95]. Another type of works study an efficient sparse data format (*e.g.,* CSR, CSB, and DIA) to reduce memory footprint and improve cache efficiency. taco [30, 53, 70] generalizes various sparse data formats with a unified expression. It generates sparse kernel code using the proper data format best fit for a class of sparsity pattern (*e.g.,* 99% sparsity). Unlike taco, SparTA proposes a holistic framework for sparsity, including sparsity propagation, execution plan transformation, and code specialization.

To optimize sparse kernels on GPU, SparseRT [79] embeds sparse weight values into kernel codes rather than stored in a sparse data format. It can be seen as a special case of code specialization in SparTA, *i.e.,* unrolling all the loops. Hong et. al [48] reorders elements in a sparse tensor and uses an adaptive tiling strategy to enhance the performance of sparse matrix multiplication. These optimizations are complementary to SparTA.

Some works [28, 88] co-design sparsity algorithms with hardware, which balance sparsity for efficient parallel execution on a GPU. Similar design has been incorporated in Sparse Tensor Core [93]. EIE [41] designs a new data encoding/decoding node and a new Processing Element (PE) to speed up matrix-vector multiplication. SCNN [65] designs another architecture of PE, which supports sparse convolution in a compressed format. SparTA can leverage these new accelerators with new transformations and specialization passes.

Sparsity exploration on DNN models. Research on both neural science and deep learning suggests that a deep neural network is sparse [54, 89]. Various model compression algorithms are shown to construct sparse models with little accuracy degradation. Unstructured pruning prunes model weights without a regular pattern [43, 54, 55], while other works prune DNN models in a regular granularity, such as in the filter [44], channel [56, 59] in CNN, and block level [61, 63]. Quantization is another way to sparsify a model, including single-precision [31,52,92], mixed-precision among layers [36, 57, 77], and mixed-precision within each tensor [66, 84]. Recent works further combine the pruning and quantization techniques [42, 74, 75, 78, 83, 90]. SparTA's TeSA abstraction could capture the sparsity patterns in all these works and generate efficient code for the sparse model.

7 Conclusion

SparTA takes a principled system approach to model sparsity in deep learning, centered on the new TeSA abstraction. SparTA is designed to accommodate a rich set of sparsity patterns, work end-to-end and across the stack to support propagation of sparsity patterns and the optimizations that take advantage of those patterns, and leverage compiler technology and hardware support, all in an extensible framework. SparTA can not only contribute to superior sparsity-induced speedup, but also accelerate model sparsity innovations within a unified framework, for the first time.

References

[1] Accelerating inference with sparsity using the nvidia ampere architecture and nvidia tensorrt. https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/, 2021.

[2] The api reference guide for cublas, the cuda basic linear algebra subroutine library. https://docs.nvidia.com/cuda/cublas/index.html, 2021.

[3] The api reference guide for cusparse, the cuda sparse matrix library. https://docs.nvidia.com/cuda/cusparse/index.html, 2021.

[4] Cloud tpu: Train and run machine learning models faster than ever before. https://cloud.google.com/tpu, 2021.

[5] Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma, 2021.

[6] cusparselt: A high-performance cuda library for sparse matrix-matrix multiplication. https://docs.nvidia.com/cuda/cusparselt/index.html, 2021.

[7] Einstein notation. https://en.wikipedia.org/wiki/Einstein_notation, 2021.

[8] Geforce rtx 2080 ti. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/, 2021.

[9] Graphcore. https://www.graphcore.ai/, 2021.

[10] Intel advanced vector extensions 512 (intel avx512). https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html, 2021.

[11] Intel fpgas and programmable devices. https://www.intel.com/content/www/us/en/products/programmable.html, 2021.

[12] Intel oneapi math kernel library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, 2021.

[13] Mixed-precision programming with cuda 8. https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/, 2021.

[14] Open neural network exchange. https://onnx.ai/, 2021.

[15] Openvino: Deploy high-performance, deep learning inference. https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html, 2021.

[16] Reproducing oopsla 2020 results. https://github.com/tensor-compiler/taco/tree/oopsla2020, 2021.

[17] Rocm sparse marshalling library. https://github.com/ROCmSoftwarePlatform/hipSPARSE, 2021.

[18] The sdk for high-performance deep learning inference. https://docs.nvidia.com/deeplearning/tensorrt/, 2021.

[19] Set cover problem. https://en.wikipedia.org/wiki/Set_cover_problem, 2021.

[20] The world's first 7nm gaming gpu. https://www.amd.com/en/products/graphics/amd-radeon-vii, 2021.

[21] Triton. https://github.com/openai/triton.git, 2021.

[22] Tvm sparsity code. https://github.com/apache/tvm/blob/254563a3140cf63fe77a46058688209de3aa213c/python/tvm/topi/cuda/sparse.py#L96, 2021.

[23] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[24] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Citeseer, 2008.

[25] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[26] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, 2009.

[27] Brad Calder, Peter Feller, Alan Eustace, et al. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1(1):1–6, 1999.

[28] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 63–72, 2019.

[29] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[30] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 823–838, 2020.

[31] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[32] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[33] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[35] Johannes Doerfert and Hal Finkel. Compiler optimizations for openmp. In *International Workshop on OpenMP*, pages 113–127. Springer, 2018.

[36] Ahmed Elthakeb, Prannoy Pilligundla, FatemehSadat Mireshghallah, Amir Yazdanbakhsh, Sicuan Gao, and Hadi Esmaeilzadeh. Releq: An automatic reinforcement learning approach for deep quantization of neural networks. In *NeurIPS ML for Systems workshop, 2018*, 2019.

[37] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[38] Jeffrey S Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *ACM SIGPLAN Notices*, 34(5):192–203, 1999.

[39] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*, 2020.

[40] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[41] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[42] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[43] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.

[44] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, 2019.

[45] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[46] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[47] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.

[48] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[49] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[50] Wei-Ning Hsu, Benjamin Bolte, Yao-Hung Hubert Tsai, Kushal Lakhotia, Ruslan Salakhutdinov, and Abdelrahman Mohamed. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *arXiv preprint arXiv:2106.07447*, 2021.

[51] Shankar Iyer, Nikhil Dandekar, Kornél Csernai, et al. First quora dataset release: Question pairs. *data. quora. com*, 2017.

[52] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[53] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[54] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[55] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

[56] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[57] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858. PMLR, 2016.

[58] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 4876–4883, 2020.

[59] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[60] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 881–897, 2020.

[61] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[62] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *arXiv preprint arXiv:1905.10650*, 2019.

[63] Sharan Narang, Eric Undersander, and Gregory Diamos. Block-sparse recurrent neural networks. *arXiv preprint arXiv:1711.02782*, 2017.

[64] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[65] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[66] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 688–698. IEEE, 2018.

[67] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[68] Ali Pinar and Michael T Heath. Improving performance of sparse matrix-vector multiplication. In *SC'99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pages 30–30. IEEE, 1999.

[69] Ram Rangan, Mark W Stephenson, Aditya Ukarande, Shyam Murthy, Virat Agarwal, and Marc Blackstein. Zeroploit: Exploiting zero valued operands in interactive gaming applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(3):1–26, 2020.

[70] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[71] Mark Stephenson and Ram Rangan. Pgz: automatic zero-value code specialization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 36–46, 2021.

[72] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020.

[73] Naftali Tishby, Fernando C Pereira, and William Bialek. The information bottleneck method. *arXiv preprint physics/0004057*, 2000.

[74] Frederick Tung and Greg Mori. Clip-q: Deep network compression learning by in-parallel pruning-quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7873–7882, 2018.

[75] Frederick Tung and Greg Mori. Deep neural network compression by in-parallel pruning-quantization. *IEEE transactions on pattern analysis and machine intelligence*, 42(3):568–579, 2018.

[76] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.

[77] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019.

[78] Ying Wang, Yadong Lu, and Tijmen Blankevoort. Differentiable joint pruning and quantization for hardware efficiency. In *European Conference on Computer Vision*, pages 259–277. Springer, 2020.

[79] Ziheng Wang. Sparsert: Accelerating unstructured sparsity on gpus for deep learning inference. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 31–42, 2020.

[80] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

[81] Chenfei Wu, Jian Liang, Lei Ji, Fan Yang, Yuejian Fang, Daxin Jiang, and Nan Duan. NÜWA: Visual synthesis pre-training for neural visual world creation. *arXiv preprint arXiv:2111.12417*, 2021.

[82] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: General and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[83] Haichuan Yang, Shupeng Gui, Yuhao Zhu, and Ji Liu. Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2178–2188, 2020.

[84] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. Mixed-precision embedding using a cache. *arXiv e-prints*, pages arXiv–2010, 2020.

[85] Shu-wen Yang, Po-Han Chi, Yung-Sung Chuang, Cheng-I Jeff Lai, Kushal Lakhotia, Yist Y Lin, Andy T Liu, Jiatong Shi, Xuankai Chang, Guan-Ting Lin, et al. Superb: Speech processing universal performance benchmark. *arXiv preprint arXiv:2105.01051*, 2021.

[86] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 285–300, 2018.

[87] Zhewei Yao, Linjian Ma, Sheng Shen, Kurt Keutzer, and Michael W Mahoney. Mlpruning: A multilevel structured pruning framework for transformer-based models. *arXiv preprint arXiv:2105.14636*, 2021.

[88] Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. Balanced sparsity for efficient dnn inference on gpu. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5676–5683, 2019.

[89] Takashi Yoshida and Kenichi Ohki. Natural images are reliably represented by sparse and variable populations

of neurons in visual cortex. *Nature communications*, 11(1):1–19, 2020.

[90] Yiren Zhao, Xitong Gao, Daniel Bates, Robert Mullins, and Cheng-Zhong Xu. Focused quantization for sparse cnns. *arXiv preprint arXiv:1903.03046*, 2019.

[91] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

[92] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[93] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–371, 2019.

[94] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

[95] Ling Zhuo and Viktor K Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74, 2005.

A Artifact Appendix

Abstract

SparTA proposes the new TeSA abstraction which enables the sparsity optimization across the compiler stack. This artifact reproduces the main results of the evaluation on NVIDIA 2080Ti and A100.

Scope

This artifact will validate the following claims:

- End-to-end performance: By reproducing the experiments of Figure 8, 9, 10, 11, we can validate the end-to-end latency and memory footprint of SparTA claimed in §5.1.
- Effectiveness of the propagation: By reproducing the experiments of Figure 14, 15, we can validate the effectiveness of the propagation.
- Effectiveness of the transformation: By reproducing the experiments of Figure 16, 17, we can validate the effectiveness of the transformation.
- Effectiveness of the specialization: By reproducing the experiments of Figure 18, we can validate the effectiveness of the specialization.
- Augmentation of model sparsity exploration: By reproducing the experiments of Figure 20, we can validate that SparTA can augment the model sparsity exploration for the algorithms.

Contents

In this artifact, we will reproduce the Figure 8-11, 14-18, 20 on NVIDIA 2080Ti and A100. Each figure has a shell script to reproduce and visualize the experimental results automatically. In addition, there are many baselines compared in our evaluation, therefore, we also provide a Dockerfile containing all dependent environments for 2080Ti and A100 respectively. Users can quickly set up the experiment environment with the Dockerfile we provided.

Hosting

The artifact is hosted at https://github.com/microsoft/SparTA/tree/sparta_artifact. To get the code, please git clone the SparTA repository and checkout to the *sparta_artifact* branch.

Requirements

- **Hardware requirements:** Figure 17 requires a NVIDIA A100 GPU and the other Figures requires a NVIDIA 2080Ti GPU.
- **Software requirements:** Please use docker to build the *image/Dockerfile* to set up the environment for 2080Ti and *image/Dockerfile.a100* to set up the environment for A100.
- **CUDA Driver:** Larger than 11.2.

Tutorial

Environment setup To set up the environment, please first clone the code and build the docker image based the Dockerfile we provided. Second, please start a docker instance and install the SparTA in the python environment. Finally, please run the *init_env.sh* to initialize the environment variables and download the datasets. Listing 1 shows the commands used to set up the experiment environment.

Listing 1: Commands to set up the environment

```
# get the source code
git clone -b sparta_artifact https://github.com/microsoft/SparTA.git
cd SparTA/image
# build the docker image
sudo docker build . -t artifact
# start a docker instance
sudo docker run -it --gpus all --shm-size 16G artifact

# Execute following commands in the docker instance
# install the sparta
mkdir workspace && cd workspace
git clone https://github.com/microsoft/SparTA && cd SparTA && git checkout sparta_artifact
conda activate artifact
python setup.py develop
# initialize the environment
cd script && bash init_env.sh
```

Run experiments SparTA provides the end-to-end scripts to reproduce all the experiments with one command on NVIDIA 2080Ti and A100 respectively. Listing 2 shows the commands to start all the experiments. The reproduced results will be visualized and saved automatically.

Listing 2: Commands to run the experiments

```
# go into the script directory
cd script
# for 2080Ti
bash run_all_2080ti.sh
# for A100
bash run_all_a100.sh
```

ROLLER: Fast and Efficient Tensor Compilation for Deep Learning

Hongyu Zhu[†◇*] Ruofan Wu[‡◇*] Yijia Diao[$◇*] Shanbin Ke[¶◇*] Haoyu Li[§◇*] Chen Zhang[£◇*]
Jilong Xue[◇] Lingxiao Ma[◇] Yuqing Xia[◇] Wei Cui[◇] Fan Yang[◇] Mao Yang[◇]
Lidong Zhou[◇] Asaf Cidon [§] Gennady Pekhimenko [†]
[†]*University of Toronto* [‡]*Renmin University of China* [$]*Shanghai Jiao Tong University*
[¶]*UCSD* [§]*Columbia University* [£]*Tsinghua University* [◇]*Microsoft Research*

Abstract

Despite recent advances in tensor compilers, it often takes hours to generate an efficient kernel for an operator, a compute-intensive sub-task in a deep neural network (DNN), on various accelerators (e.g., GPUs). This significantly slows down DNN development cycles and incurs heavy burdens on the development of general kernel libraries and custom kernels, especially for new hardware vendors. The slow compilation process is due to the large search space formulated by existing DNN compilers, which have to use machine learning algorithms to find good solutions.

In this paper, we present ROLLER, which takes a different construction-based approach to generate kernels. At the core of ROLLER is *r*Tile, a new tile abstraction that encapsulates tensor shapes that *align* with the key features of the underlying accelerator, thus achieving efficient execution by limiting the shape choices. ROLLER then adopts a recursive *r*Tile-based construction algorithm to generate *r*Tile-based programs (*r*Program), whose performance can be evaluated efficiently with a micro-performance model without being evaluated in a real device. As a result, ROLLER can generate efficient kernels *in seconds*, with comparable performance to the state-of-the-art solutions on popular accelerators like GPUs, while offering better kernels on newer accelerators like IPUs.

1 Introduction

Deep neural networks (DNN) have been used extensively in intelligent tasks like computer vision and natural language understanding. As DNN computation is known for its complexity, the compute intensive sub-tasks (e.g., matrix multiplication) in a DNN model are abstracted as operators and implemented as kernels, executed on modern accelerators (e.g., GPUs, TPUs) to speed up the computation. DNN compilers play an important role in producing high-performance kernels for the development of DNN models. It reduces the burden of (often hand-crafted) library-based kernel development (e.g., cuDNN [6] and cuBLAS [2]) and provides a flexible way to cover the fast-growing number of custom operators, which libraries struggle to catch up with and optimize, a growing pain especially for new hardware vendors.

DNN compilers treat a DNN operator as tensor computation, which is then translated into nested multi-level loops iterated over the computation on each tensor element along different axes (dimensions). Compiler optimization techniques like loop partitioning/fusion/reordering are applied to nested loops. Due to the inherent complexity of loop rearrangement, it is a combinatorial optimization problem to find a good solution among a large search space, often with millions of choices. Therefore, advanced compilers [15, 33, 35] propose to adopt machine learning algorithms to search for a good solution. This usually takes thousands of search steps, each evaluated in a real accelerator, to find a reasonable solution. Our own experience shows that tuning an end-to-end DNN model using state-of-the-art compilers [15, 33] often requires days, if not weeks. The tuning time may be even longer if the DNN model runs on less mature accelerators (e.g., AMD GPU or Graphcore IPU [4]) (§2). To make the matter worse, a DNN model need to re-compile whenever its structure, operator types, tensor shapes and configurations are changed. This is often required when trying different configurations in model training or inference. Given that an operator could have arbitrary input shapes and configurations, such compilation could significantly slow down the overall DNN model development cycle.

In this paper, we propose ROLLER, a deep learning tensor compiler that addresses the problem in a radically different way. ROLLER is built on the following insights. First, instead of multi-level nested loops, ROLLER treats the computation in a DNN operator as a *data processing pipeline*, where data tiles (a fraction of a tensor) are moved and processed in an abstracted hardware with parallel execution units and multi-layer memory hierarchy. The goal of generating efficient kernel programs then becomes that of improving the throughput of the pipeline.

*Work is done during the internship at Microsoft Research.

Second, for an accelerator to execute efficiently, the shape of a data tile should *align* with the hardware characteristics, including memory bank, memory transaction length, and minimum schedulable unit (e.g., warp size in GPUs). To achieve the full alignment across multiple hardware features, the available tile shapes are limited. More importantly, with alignment as a constraint, to maximize the throughput of a pipeline, one only needs to *construct* an aligned tile shape that saturates the execution unit of the accelerator. This construction process is significantly more efficient than solving the original unconstrained combinatorial optimization problem.

Third, the performance of an aligned pipeline is highly predictable. Key performance metrics under the aligned pipeline (e.g., memory throughput) can be derived from the hardware specification (or through micro-benchmarking). This greatly simplifies the performance evaluation under various aligned configurations, eliminating the need of a complex cost model and/or expensive hardware-based evaluation on each aligned configuration.

With these insights, ROLLER proposes *r*Tile, a new abstraction that encapsulates data tile shapes that *align* with the key features of the hardware accelerator and the input tensor shapes (§3.1). A data processing pipeline can then be described as an *r*Tile-based program (a.k.a. *r*Program) composed by three interfaces: `Load`, `Store`, and `Compute`, acted against *r*Tile. To construct an efficient *r*Program, ROLLER follows a *scale-up-then-scale-out* approach. It first performs the scale-up process, which adopts a recursive *r*Tile-based construction algorithm (Figure 8) to gradually increase the size of the *r*Tile shape to construct an *r*Program that saturates a single execution unit of the accelerator (e.g., an SM, a streaming multi-processor in a NVIDIA GPU). It then performs the scale-out process, which simply replicates the resulting *r*Program to other parallel execution units, thanks to the homogeneity of both the computation pattern of deep learning and the parallel execution units in an accelerator.

ROLLER can evaluate the performance of different *r*Tiles without significant overheads. The peak (saturate) compute throughput can simply be measured *once per operator type*. And due to the alignment, other key performance factors like memory pressure of an *r*Tile can be derived analytically from hardware specifications. This leads to an efficient micro-performance model, avoiding the expensive online profiling on each configuration required by existing DNN compilers, thereby significantly speeding up the compilation process. In addition, due to the strict alignment requirements, the recursive construction process can produce a few desired *r*Tiles (and *r*Program) quickly. Combined, ROLLER can generate efficient kernels *in seconds*.

We have implemented ROLLER on top of TVM [15] and Rammer [26], and open-sourced the code[1]. Our evaluation on 6 types and 119 popular DNN operators from several mainstream DNN models shows that ROLLER can generate highly-optimized kernels in *seconds*, especially for large expensive custom operators. This achieves *three orders of magnitude improvement* on compilation time. The performance of ROLLER-generated kernels is comparable to and often better than the state-of-the-art tensor compilers and even vendor-provided DNN libraries. With the three *r*Tile-based interfaces (`Load`, `Compute`, `Store`) describing an *r*Program, ROLLER can easily adapt to different accelerators like AMD GPU and Graphcore IPU. ROLLER has been used to develop custom DNN kernels internally and shown to significantly speed up our development cycle. It offers potentially disruptive opportunities to new players in the compute accelerator market, who previously have to spend significant engineering efforts on efficient kernels.

[1] https://github.com/microsoft/nnfusion/tree/osdi22_artifact/artifacts

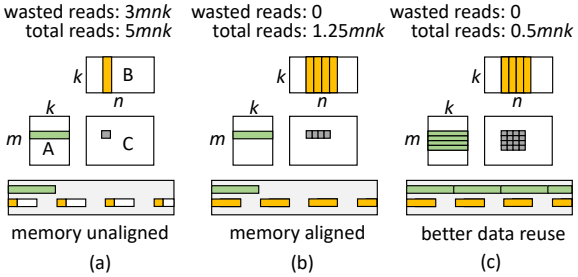


Figure 1: Access pattern of different tile shape. Matrix multiplication, $C_{m,n} = A_{m,k} \times B_{k,n}$.

2 Motivation and Key Observations

Excessive compilation time. Our own experience in a set of DNN operators (detailed setting in §5) shows that the average compile time for a single operator using Ansor [33], a state-of-the-art tensor compiler, is 0.65 hours. Among them, one convolution operator in ResNet model takes 2.17 hours. A DNN model may contain hundreds of operators, thus it easily takes days to compile the model. For example, to compile a NASNet model (§5), we reach only 32% of the overall searching progress after tuning for 41.8 hours. Our experience also shows the compilation speed is even worse on less mature devices, the compiler takes much longer time for a kernel.

Observation and insights. We observe that there exists a different view to the computation of an DNN operator. Taking matrix multiplication (MatMul), $C_{m,n} = A_{m,k} \times B_{k,n}$, as an example to illustrate our observation. Unlike existing compilers that treat MatMul as a 3-level loop iterated over each axis m,k,n, the computation process is also a data processing pipeline. One can `Load` each sub-matrix (i.e., a tile) from A and B, `Compute` the two tiles, and `Store` the resulting tile of C to memory. Thus, the performance of the computation depends on how fast one can move the data tiles in the `Load-Compute-Store` pipeline.

The key factor affecting the performance in all steps in the pipeline is the *shape* of tiles and the corresponding layout

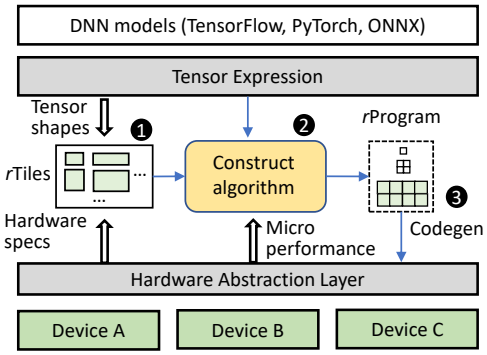


Figure 2: System overview of ROLLER.

in the one-dimension memory space. Figure 1(a) illustrates the computation of one element in C (in the top part) and the memory accessing pattern (in the bottom part). Assuming all matrices stored in a row-major layout, loading a column from B causes strided accesses in length of 1. Suppose the memory transaction length is 4, there will be $3/4$ of total redundant data reads. Thus, the data tile shape should align with the memory transaction length for efficient memory access. In Figure 1(b), when computing B in the granularity of 1×4 tile, there will be no memory bandwidth waste. Besides memory alignment, the tile shape should also align with the hardware execution unit, e.g., the parallel threads number, to avoid waste in computing cycles. Moreover, the tile shape also affects data reuse opportunities due to caching, a common feature in modern accelerators. For example, Figure 1(a) needs $2mnk$ data reads when computing a 1×1 tile each time. However, in Figure 1(b), only $1.25mnk$ reads are required, as one read from A can be reused 4 times. If setting the tile size along M dimension to 4×4, as shown in Figure 1(c), the total reads can be reduced to $0.5mnk$. A $10\times$ improvement over Figure 1(a).

These observations motivate ROLLER, a system that identifies the aligned tile shapes and constructs an efficient tile processing pipeline to improve the end-to-end throughput.

3 System Design

Figure 2 shows the system overview. ROLLER takes an operator described as a tensor expression (§3.1). The expression is generated by users or from a graph-level DNN compiler [15, 26, 33], which might further fuse multiple operators into a single expression. ROLLER extracts the tensor shapes from the tensor expression and leverage hardware specifications to construct *r*Tiles, i.e., a hardware-aligned building block (§3.1). Based on *r*Tiles, ROLLER proposes a *scale-up-then-scale-out* recursive construction algorithm to generate efficient tensor programs (named *r*Program) that describes the data processing pipeline (§3.2). When generating *r*Program, the construction algorithm identifies good *r*Tile configurations by evaluating the performance of a constructed *r*Program through a micro-performance model. It is built on top a device described through a hardware abstraction layer exposing only *r*Tile-related interfaces: `Load`, `Compute`, and `Store` (§3.3). The constructed *r*Program is finally realized through a code generator to emit the final kernel code corresponding to the specific device.

```
class rTile {
  TensorExpr expr;
  TileShape shape;
  TileShape storage_padding;
  vector<TileShape> GetInputDataTiles();
  vector<TileShape> GetOutputDataTiles();
};
```

Figure 3: The data structure of *r*Tile.

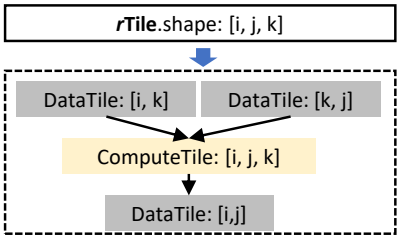


Figure 4: The data tiles and computing tile inferred by an *r*Tile for MatMul expression.

3.1 Tensor Expression and *r*Tile

ROLLER takes input of a tensor computation as an index-based lambda expression, i.e., tensor expression [15, 27]. It describes how each element in the output tensor is computed based on the corresponding elements in the input tensors. For example, a MatMul operator with output tensor C of the shape $M \times N$ can be expressed as,

```
C = compute((M,N), lambda i,j:sum(A[i,k]*B[k,j])),
```

where the element indexed by (i, j) in C is computed by a sum reduction over the elements in row i of A and column j of B, and k is the reduction axis. Such an expression can cover the majority of operators in DNN models and is widely used in existing DNN compilers including TVM [15], Ansor [33], and FlexTensor [35].

ROLLER introduces *RollingTile* (*r*Tile for short) as the basic computing unit to compose a tensor computation. As shown in Figure 3, an *r*Tile encapsulates a multi-dimensional tile `shape` defined along each loop axis of a given tensor expression `expr`. Given `shape` and `expr`, an *r*Tile can statically infer the involved input and output data tiles. For example, a tile shape $[4, 4, 2]$ along axes i, j, k denotes an *r*Tile for the above MatMul expression, where each *r*Tile loads a 4×2 data tile from A and a 2×4 tile from B, conducts total $4 \times 4 \times 2$ multiply-add computations, and stores a 4×4 data tile to C, as illustrated in Figure 4.

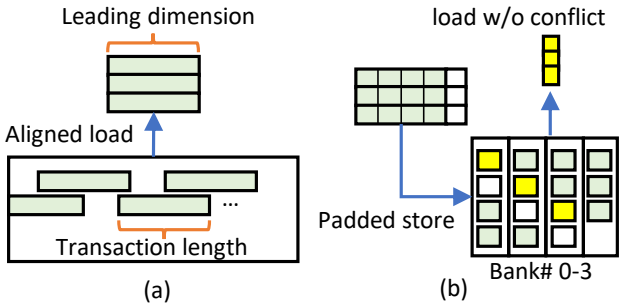


Figure 5: Illustration of (a) transaction aligned memory load and (b) bank conflict-free padding.

A unique property of an *r*Tile is that it must align with both the underlying hardware features and the tensor shapes in a given tensor expression. All these alignments are controlled by the *r*Tile *shape* and the *storage_padding* fields in Figure 3, which represent the logical form and the physical layout of an *r*Tile, respectively. We elaborate the detailed requirements of alignment next.

Alignment with the hardware execution unit. First, the shape of an *r*Tile must align with the parallelism of the execution unit it runs on. For example, if running on a warp of threads in a GPU, the size of `shape` in the *r*Tile should be a multiple of the warp size, e.g., 32, for maximal computing efficiency. When using TensorCore in NVIDIA GPUs, the *r*Tile shape should be a multiple of $16 \times 16 \times 16$. Similarly, an *r*Tile executed on a streaming multi-processor (SM) should align its size as a factor of execution unit number on the SM.

Alignment with memory transaction. Second, a data tile's shape should align with the length of memory transaction for optimal memory access. Specifically, for each data tile of an *r*Tile, we should guarantee that its leading dimension (e.g., the inner-most dimension in a row-major tensor) is a multiple of the memory transaction length, as illustrated in Figure 5(a). In ROLLER, tensors are allocated in a cache-aligned way. Thus, an *r*Tile can avoid any wasted transaction read, as its shape is aligned with the memory transaction.

Alignment with memory bank. Third, the memory layout of a data tile should align its stride with the memory bank to avoid read conflicts. For example, a $[3,4]$ data tile is kept in the memory across 4 banks and is read by an upper-memory-layer tile with a shape of $[3,1]$, as shown in Figure 5(b). A naive approach that stores all the $[3,1]$ values in the same bank will result in conflicted loading. *r*Tile avoids such inefficiency by padding a data tile. Given a data tile with a leading dimension of size N, which is read by another tile with a leading dimension of size n, we add a padding size of $(BL - N\%(BL) + L\lceil n/L \rceil)\%(BL)$ along N when storing this tile, where B and L are the bank number and the bank width, respectively. The padding sizes along each axis are calculated and stored in the `storage_padding` field in Figure 3. For the case in Figure 5(b), by a padding size of 1, all the $[3,1]$ values are distributed in different banks and can be read efficiently.

Alignment with tensor shape. Finally, an *r*Tile's shape should align with the tensor shape of an input tensor expression. Otherwise, the computation cannot be evenly partitioned by the *r*Tile, wasting compute resources or incurring heavy boundary checking overheads. A simple solution is to add a padding size P_i along a tensor dimension i with size of N_i, which makes $N_i + P_i$ a multiple of the *r*Tile shape's dimension size at axis i. However, a large padding might waste computation. ROLLER therefore restricts tensor padding under a range ε, where an *r*Tile's shape dimension size S_i has to satisfy that $\frac{S_i - N_i \% S_i}{N_i} \leq \varepsilon$, where N_i is the tensor size at dimension i. This ensures the wasted percentage of computation is bounded by ε. With this restriction, we can enumerate all the valid *r*Tile shapes that satisfy this condition.

Deriving all *r*Tiles. Given the above alignment requirements, for a specific tensor expression and hardware device, ROLLER incrementally derives all the conforming *r*Tiles using the following interface:

```
vector<int> GetNextAlignedAxisSize(rTile T, Dev d),
```

which returns the next aligned size for each axis in the `shape` of *r*Tile T given the specific device specification d. This is calculated by gradually increasing the dimension size along each axis until it satisfies all the alignment requirements. The *r*Tile abstraction allows ROLLER to be extended to support new alignment requirements (e.g., new hardware features). This is achieved by adding new alignment rules to the `GetNextAlignedAxisSize` interface.

Calculating data reuse score. An interesting property of *r*Tile is that we can implicitly control the memory traffic by adjusting its `shape`. Increasing the *r*Tile size usually brings more *data reuse* opportunities at the cost of occupying more memory space. Given an *r*Tile T and its next aligned size in each axis, we can calculate the data reuse score S_i for axis i by $S_i = \frac{Q(T) - Q(T_i')}{F(T_i') - F(T)}$, where T_i' is a newly enlarged *r*Tile by replacing the dimension size at axis i with the next aligned size from `GetNextAlignedAxisSize`. Functions $Q(T)$ and $F(T)$ calculate the memory traffic and memory footprint when the computation is executed in the granularity of T, which can be directly inferred based on the given tensor expression and hardware memory specification (§3.3). A larger S_i means better cost-efficiency, i.e., more memory traffic can be saved with the same memory usage. The memory reuse score plays a critical role in constructing an efficient *r*Program (using *r*Tiles), as shown in the next subsection.

3.2 Tensor Program Construction

***r*Tile program.** Given *r*Tile and the hierarchical memory structure of modern accelerators, a tensor computation can be naturally treated as a streaming data processing pipeline. The computation loads data tiles (specified in *r*Tile) from the lowest memory layer through the memory hierarchy to the highest layer, performs *r*Tile computation on the execution

```
for L1_iter in L2_rtile.split(L1_rtile):
  L1_input_tiles = Load(L1_iter); //L2 to L1
  for L0_iter in L1_rtile.split(L0_rtile):
    L0_input_tiles = Load(L0_iter) //L1 to L0
    L0_out_tile = Compute(L0_input_tiles);
    Store(L0_out_tile, L2_out_tile);//L0 to L2
```

Figure 6: The pseudo code of an *r*Program on a device with a 3-layer memory hierarchy (Bottom-up: layer `L2` to layer `L0`).

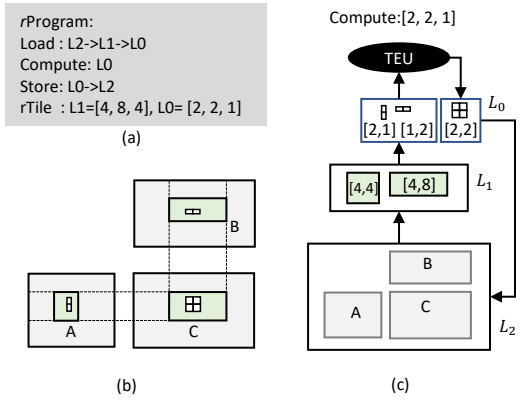


Figure 7: ROLLER computation model. (a) An *r*Tile program; (b) *r*Tiles on matrix multiplication; (c) Execution of the *r*Tile program on a hardware memory hierarchy.

units of the accelerator, and stores the resulting data tiles back to the lowest memory. For each memory layer, a specific *r*Tile is defined to align with the characteristics of this memory layer. Thus, ROLLER describes tensor computation as a data processing pipeline with a hierarchical *r*Tile configuration, which is called an *r*Tile program (i.e., *r*Program).

Figure 6 shows an *r*Program on a device with three memory layers (L0, L1 and L2). The *r*Program is described by the *r*Tile at each layer and the *r*Tile instructions (i.e., `Load`, `Store`, and `Compute`) at each memory layer. Figure 7(a) shows a MatMul *r*Program illustrated in Figure 7(b). Figure 7(c) illustrates how the *r*Program is mapped to each memory layer of a device. Specifically, each time it loads a $[4,4]$ data tile in A and a $[4,8]$ tile in B from memory L2 to L1; and then it loads the data tiles from memory L1 to memory L0 (i.e., registers) in shapes of $[2,1]$ and $[1,2]$. After each `Compute`, the resulting $[2,2]$ tile will be directly stored from L0 to L2.

Given a data processing pipeline, the optimization goal of the corresponding *r*Program is to maximize the throughput of the pipeline. The goal can be translated into three conditions: 1) the computation and memory movement should fully leverage the hardware features; 2) the throughput should saturate the bottleneck stage; and 3) there needs to be sufficient parallelism to leverage all the parallel execution units. Thus, ROLLER proposes the following *r*Program construction policy: first scale-up on one core by constructing a single-core *r*Program to saturate the core's hardware utilization and then

```
1  Func ConstructProg(expr:TensorExpr, dev:Device):
2      T = rTile(expr);
3      Results = [];
4      EnlargeTile(T, dev.MemLayer(0), rProg());
5  Func EnlargeTile(T:rTile, mem:MemLayer, P:rProg):
6      if mem.IsLowestLayer()
7          Results.append(P);
8          if (Results.Size() > TopK) Exit();
9          Return();
10     for T' : GetNextRTileShapes(T, mem) do
11         if Visited(T')
12             Return();
13         if MemFootprint(T') > mem.Capacity()
14             P.Add(mem, T);
15             EnlargeTile(T, mem.Next(), P);
16         else
17             if MemPerf(T') > MaxComputePerf(T'.expr)
18                 P.Add(mem, T');
19                 EnlargeTile(T', mem.Next(), P);
20             EnlargeTile(T', mem, P);
21 Func GetNextRTileShapes(T:rTile, mem:MemLayer)
22     alignedSizes = GetNextAlignedAxisSize(T, mem);
23     SortedRTiles = OrderedMap();
24     for d : T.Dimensions() do
25         T' = T.Replace(d, alignedSizes[d]);
26         SortedRTiles.Insert({T', DataReuseScore(T')});
27     Return SortedRTiles;
```

Figure 8: ROLLER's *r*Program constructing algorithm for a single core (e.g., an SM).

scale-out to leverage the multi-core parallelism by replicating the constructed single-core *r*Program.

Scaling up an *r*Program. Since the alignment properties of *r*Tile ensure hardware efficiency, ROLLER can just focus on maximizing the throughput at each memory layer by constructing the right *r*Tile shape. By leveraging the data reuse score defined in §3.1, the single-core *r*Program construction algorithm starts from an initial *r*Tile and gradually enlarges it towards the most cost-effective axis in the *r*Tile (i.e., with the maximum data reuse score). Note that the construction algorithm does not require an absolute data reuse score, it just picks the largest one to maximize the throughput. During the process, the memory performance improves until it hits the computational bound or the maximal memory capacity. The above process repeats for each memory layer from top to bottom, until a desired *r*Program is constructed. Note that if the data reuse score remains constant for some tensor expressions, e.g., element-wise operators, ROLLER will just construct *r*Tiles for the top layer and loads them directly from the bottom layer memory.

Figure 8 shows the detailed construction algorithm. Given a tensor expression `expr` and a target device `dev`, the algo-

rithm constructs an initial *r*Tile T at the top memory layer and enlarges T recursively (`EnlargeTile` in line 4). At each step, it enumerates the next larger *r*Tile T' that improves the data reuse score most (`GetNextRTileShapes` in line 10). If T' hits the memory capacity (line 13) or the data tile loading throughput $MemPerf(T')$ exceeds the peak computing throughput $MaxComputePerf(T')$ (line 17), the algorithm records the current *r*Tile T and goes on to `EnlargeTile` at the next memory layer. Otherwise, it continues to enlarge T' at the current layer (line 20). The construction finishes at the lowest memory layer (line 6), producing one result and repeating, until it obtains K (e.g., 5-20) *r*Programs (to tolerate the hidden factors affected by the device compiler). Note that $MemPerf(T')$ and $MaxComputePerf(T')$ are derived based on `dev`, based on the micro-performance model (§3.3).

Scaling out an *r*Program. Given the homogeneity of both the computation pattern of most DNN operators and the parallel execution units in an accelerator, ROLLER simply replicates the *r*Program constructed on one execution unit to other units, by uniformly partitioning the computation into *r*Tiles of the size equals to the lowest layer *r*Tile. We achieve this by distributing all the partitions evenly to all execution units. Note that ROLLER prefers to assign the partitions split along a reduction axis on the same execution unit, as they can share the reduction results in the higher memory layers. Note that ROLLER does not assume an *r*Program will exclusively occupy all computing units, the system can explicitly control the parallelism of a *r*Program when scaling out.

Small operator and irregular tensor shape. The scale-out algorithm inherently favors operators with sufficient parallelism, e.g., where the partition number is significantly larger than the number of execution units. For a small operator, the overall performance of the algorithm could suffer from the low utilization of parallel execution units. In general, this can be addressed by co-scheduling with other operators in compilers like Rammer [26], if there exists sufficient inter-operator parallelism. Otherwise, for each *r*Program, ROLLER will try to shrink its *r*Tiles along the axis that has the smallest data reuse score to achieve sufficient parallelism. Note that this enumerating process returns the next aligned tile size each time just like other alignment rules, which is an efficient process and incurs negligible costs compared to the overall construction process.

In addition, a large operator may contain irregular tensor shapes with small dimensions, whereas ROLLER might not generate a sufficient number of *r*Programs due to the alignment requirements. To address this issue, ROLLER transforms a tensor expression into a canonical form by an axis fusion pass. Specifically, for all the involved tensors, if there exist two adjacent axes in one tensor, which are either both existing and still adjacent or both missing in all other tensors, ROLLER can safely merge these two axes. For example, an element-wise operator with the tensor shape $[17, 11, 3]$ in both input and output tensors, ROLLER will transform it into the tensor shape $[561](17 \times 11 \times 3)$ by fusing the three axes. Besides axis fusion, ROLLER will also try to greedily increase the parameter ε in the tensor padding mechanism (§3.1) until K *r*Programs have been constructed.

```
// compute interface
int Load(T* src, rTile st, T* dst, rTile dt);
int Store(T* dst, rTile dt, T* src, rTile st);
int Compute(TensorExpr e, rTile t, T** args);

Spec GetDeviceSpec(); // Spec query interface

// interfaces of the micro-performance model
size_t MemFootprint(rTile t);
size_t MemTraffic(rTile t);
double MaxComputePerf(TensorExpr expr);
double MemPerf(rTile t);
```

Figure 9: The interface of ROLLER's hardware abstraction

3.3 Efficient Evaluation of an *r*Program

In the construction algorithm, ROLLER needs to evaluate the performance of *r*Program. Instead of evaluating the end-to-end *r*Program in a real hardware device, ROLLER only needs to evaluates the performance of the corresponding *r*Tile, e.g., `MemPerf` and `MaxComputePerf` in Figure 8.

To this end, ROLLER builds a micro-performance model against a device described in a hardware abstraction layer (HAL). The HAL models an accelerator as multiple parallel execution units with a hierarchical memory layer. The HAL exposes three *r*Tile-based interfaces: `Load`, `Compute`, and `Store` (Figure 9). An execution unit is abstracted as an *rTile Execution Unit (TEU)*, which computes the data tiles through the `Compute` interface. Multiple TEUs can be organized as a group, which `Load` and `Store` tiles cooperatively. The HAL treats different memory layers, e.g., register, shared memory, DRAM, as an unified type exposing the hardware specifications that affect the performance of tile movement. The specifications include memory capacity, transaction lengths, cache line size, and number of memory banks, which can be obtained by the `GetDeviceSpec` interface in Figure 9.

Micro performance model. With the hardware abstraction layer, ROLLER can easily derive the performance of a *r*Tile (and hence the *r*Program). First, given an *r*Tile, the incurred memory footprint (including padding) and the memory traffic volume across different layer can be statically inferred from the *r*Tile's tensor expression `expr` and the `shape`, i.e., the `MemFootprint` and `MemTraffic` interfaces in Figure 9. They are used to calculate the data reuse scores and check if an *r*Tile exceeds the memory capacity. Second, to calculate `MaxComputePerf` of an *r*Tile, ROLLER conducts a one-time profiling to measure the peak compute throughput by aggressively enlarging the compute tiles (e.g., multiple of warp size in an SM) to saturate the TEU. This performance data is cached in ROLLER for future query in the construction al-

gorithm. Finally, for a given *r*Tile, ROLLER also estimates `MemPerf`, the performance on loading data tiles from a memory layer to a higher layer. Given the aligned memory access in *r*Tile, the latency of loading a regular chunk of data can be simply modeled by the division of the total traffic to the memory bandwidth. For the memory layer shared by all TEUs, we split the bandwidth evenly. For the smaller accessing sizes, ROLLER also conducts a one-time offline profiling for each device type and cache the results. It is worth noting that the micro-performance model only *needs* to be accurate when the tile shapes are fully aligned, a key requirement of ROLLER.

4 Implementation

Our implementation of ROLLER is based on TVM [15] and Rammer [26], two open-source DNN compilers. ROLLER's core mechanisms, including expression optimization, construction algorithm, micro-performance model, etc., are implemented with 8K lines of code. ROLLER's compilation pipeline is as follows. Its input is an ONNX graph [9] or a TensorFlow frozen graph [13]. ROLLER first leverages Rammer to conduct graph level optimizations (e.g., inter- and intra-operator co-scheduling). Next ROLLER derives the TVM tensor expressions for each (fused) operator extracted from the optimized graph, and generates corresponding *r*Program by ROLLER's construction algorithm, and performs kernel generation. Finally, the generated kernels are injected to Rammer's runtime and generate the end-to-end model code.

Code generation. Given the fixed code structure in an *r*Program (in Figure 6), ROLLER generates the kernel code through a predefined template, implemented as a TVM schedule with its built-in scheduling primitives. Loading and storing data tiles at each memory layer are implemented by TVM's `cache_read` and `cache_write` primitives. Partitioning on *r*Tile is done through `split` and `fuse`. Some primitive *r*Tile computation is implemented with TVM's intrinsic API. With the template, a given *r*Program can be directly generated into device codes, e.g., CUDA kernels.

Tensor padding. ROLLER relies on tensor padding to align *r*Tiles with tensor shape. In practice, most tensors in the lowest memory (e.g., DRAM) are allocated by external program (e.g., DNN framework), thus we just apply padding in the upper layer memory (e.g., shared memory). Our tensor padding currently requires the input tensor expression to specify whether it allows to pad, as well as the default padding value (e.g., 0 for MatMul operator). For the storage padding for memory bank alignment, we leverage TVM's `storage_align` primitive to add padding.

Performance profiling. ROLLER implements two profilers: a *micro-performance profiler* and a *kernel profiler*. The former generates device specifications, e.g., memory bandwidth, computing throughput, etc., through a set of micro-benchmarks, which is a one-time offline profiling for each device type and tensor expression types (regardless of the tensor shapes). The latter profiles the fastest kernels among the top K *r*Programs and is used for each compilation result if the K is larger than 1. In practice, the performance of a specific kernel code is also slightly affected by some device-compiler and hardware related hidden factors, which ROLLER can hardly control. These factors include instruction density of different instruction types, register allocation behaviors, device compiler optimizations, warp scheduling overhead, etc. Particularly, on NVIDIA GPUs, ROLLER relies on nvcc [3] to compile the generated CUDA codes into machine code. However, nvcc's proprietary optimizations might undesirably affect the program execution behaviors. Thus, ROLLER leverages the kernel profiler to quickly evaluate top performing *r*Programs and select the best one. A larger K could generally increase kernel quality. After evaluating the top 10, 20, and 50 results, our experiences show that top 10 could recall the optimal results for most cases. Note that ROLLER's kernel profiler differs from the evaluation process driven by a machine learning algorithm in previous compilers [15,33,35]. The ML-based approach usually requires hundreds even thousands of *sequential* evaluation steps, while ROLLER only profiles tens of candidates *in parallel*. In future, we plan to implement assembly-level code generation to alleviate the hidden issues in a high-level device compiler.

ROLLER's HAL allows us to support different accelerators easily. User can configure the corresponding HAL for each device type. ROLLER also provides built-in configurations for most common device types. Some detailed configurations, e.g., memory bandwidth, rely on micro-benchmark profiling or derive from published device specifications. Next, we share our experiences in implementing the HAL on several popular DNN accelerators, including NVIDIA GPUs, AMD GPUs and Graphcore IPU.

ROLLER on NVIDIA CUDA GPUs. An NVIDIA GPU usually employs a centralized memory architecture. We implement ROLLER on V100 and K80, two CUDA GPUs with different architectures on the *streaming multi-processors (SMs)*. Their memory architecture contains global memory, L2 cache, L1 cache, shared memory, and register. In ROLLER's HAL, we abstract them into 3 memory layers: L2 layer for global memory and L2 cache, L1 layer for only the shared memory, and the L0 layer for register. We ignore L1 cache because it shares the space with shared memory and cannot be controlled by user programs. The memory bandwidths of all levels are measured by our micro-benchmarks. The transaction length at the global memory layer is set to 32 Bytes, i.e., 8 float elements, for both GPUs. For V100 GPUs, the bank number and the bank length of the shared memory is 32 and 4 Bytes respectively. For K80 GPUs, the bank length is 8 Bytes. The shared memory capacities are set as 48KB for both GPUs (based on `deviceQuery`).

We implement the TEU on CUDA GPUs as a warp of 32 threads, which is also the basic unit to execute the TensorCore WMMA instructions. The size of a TEU Group on a HAL

(e.g., a SM) is set to the warp scheduler number, which is 4 for both GPUs. The SM number is 80 for V100 [21] and 13 for K80. On CUDA GPUs, each thread has a limited register capacity, e.g., 255 registers for V100. Exceeding this limit will lead to register spilling, causing significant performance degradation. This sets a limit to the size of an *r*Tile at register layer. We notice that the *nvcc* compiler will implicitly declare more registers (for loop variables or other purposes). Given that this behaviour is hard to predict, we reduce the register limit empirically to only 96 registers for both V100 and K80 per thread to avoid unexpected performance impacts.

ROLLER on AMD ROCm GPUs. We also implement ROLLER on MI50 [12], AMD's second-generation Vega series GPU. MI50 shares a similar memory architecture as V100: the centralized global memory can be accessed by all *compute units (CUs)*. Like SMs in NVIDIA GPU, each CU has its own scratchpad memory, registers, and computation cores. The data movement of a ROCm [1] kernel program is also similar. The memory transaction size for the global memory is set as 64 Bytes. The memory bank number is 32 and bank length is also 4 Bytes. We also implement the TEU as a warp of threads, which is 64 threads on MI50 GPUs. The maximal register size is empirically limited to 70 registers per thread. All other specifications such as the memory bandwidths at each layer, peak computing throughput, etc., are measured with our micro-benchmark.

ROLLER on Graphcore IPUs The Graphcore IPU [22] is a massive parallel MIMD processor with 1216 parallel processing cores. Distinct from NVIDIA and AMD GPUs, an IPU employs a distributed memory architecture. There is only 256KB on-chip local memory attached per core, and no unified global memory. When the local memory is unable to hold all the input data, by default, the initial data of a kernel program is stashed in the on-chip local memory and evenly distributed across the nodes. Thus, ROLLER's HAL for IPUs also abstracts three memory layers: L2 for all the remote memories across all cores, L2 for the local memory on each core, and L0 for the register. We take advantage of prior benchmarking work [22], which has successfully measured peak memory bandwidth and computation throughput. The size of the register files per IPU core is not publicly available. Considering that we have no prediction for behaviours of the IPU program compiler, we allow each upper-level *r*Tile to use only 10 registers, which safely guarantee that the tiling algorithm does not emit invalid tiling configurations.

5 Evaluation

We evaluate ROLLER on both DNN operator benchmarks and end-to-end models by comparing with state-of-the-art DNN compilers and frameworks. We first summarize our findings: 1) ROLLER achieves *three orders of magnitude speedup* on compilation time, compared to TVM and Ansor. On V100 GPU, the most expensive operator takes 43 seconds, while all other operators take only around 13 seconds to compile. 2) ROLLER matches the state-of-the-art performance of vendor libraries and other compilers on a wide range of operators. It even outperforms others for more than 50% of operators. 3) For operators with smaller sizes and irregular shapes, ROLLER's results are sub-optimal because of the difficulty in aligning with the hardware. However, their kernel execution time is usually small (around or below 1ms). 4) We have conducted the most extensive evaluations (119 ops in total) covering different operator types over different accelerators.

Operator	Configuration	Note
MatMul	M=65536,K=2,N=1024	M0
MatMul	M=128,K=4032,N=1000	M1
MatMul	M=65536,K=1024,N=4096	M2
Conv2D	D=(128,128,28,28), K=(128,128,3,3),S=1	C0
Conv2D	D=(128,128,58,58), K=(128,128,3,3),S=2	C1
Conv2D	D=(128,256,30,30), K=(256,256,3,3),S=2	C2
DepthwiseConv	D=(128,84,83,83), K=(84,84,5,5),S=2	D0
DepthwiseConv	D=(128,42,83,83), K=(42,42,5,5),S=1	D1
DepthwiseConv	D=(128,84,21,21), K=(336,336,1,1),S=1	D2
Element(Relu)	I=(128,1008,42,42)	E0
Element(Relu)	I=(128,256,14,14)	E1
Element(Relu)	I=(128,1024,14,14)	E2
Avgpool	D=(128,168,83,83),K=1,S=2,VALID	P0
Avgpool	D=(128,617,21,21),K=3,S=2,SAME	P1
Avgpool	D=(128,42,83,83),K=3,S=1,SAME	P2
ReduceMean	I=(128, 512, 1024), axis=[2]	R0
ReduceMean	I=(65536, 1024),axis=[1]	R1
ReduceMean	I=(128, 4032, 11, 11), axis=[2,3]	R2

Table 1: A subset of operator configurations in our benchmark.

Experimental setup. ROLLER is evaluated on four types of servers equipped with different accelerators. The CUDA GPU evaluations use two types of servers: an Azure NC24s_v3 VM equipped with Intel Xeon E5-2690v4 CPUs and 4 NVIDIA Tesla V100 (16GB) GPUs and an Azure NC24_v1 VM with 24 Intel(R) Xeon(R) CPU E5-2690v3 CPUs and 4 NVIDIA Tesla K80 GPUs. Both running on Ubuntu 16.04 with CUDA 10.2 and cuDNN 7.6.5. The AMD ROCm GPU evaluations use a server equipped with Intel Xeon CPU E5-2640 v4 CPU and 4 AMD Radeon Instinct MI50 (16GB) GPUs, installed with Ubuntu 18.04 and ROCm 4.0.1 [1]. The IPU evaluations use an Azure ND40s_v3 VM equipped with Intel Xeon Platinum 8168 CPUs and 16 IPUs with Poplar-sdk 1.0.

We compare ROLLER against other tensor compilers, vendor libraries and DNN frameworks, including TVM [15] (v0.8) and Ansor [33] (v0.8), two state-of-the-art tensor compilers; cuDNN, cuBLAS, rocBLAS (ROCm GPUs), POPLAR library (Graphcore IPU), which are vendor libraries; TensorFlow (v1.15), a state-of-the-art DNN framework; TensorFlow-XLA a state-of-the-art DNN full-model compilers; and TensorRT (v7.0) (with TensorFlow integration version), a vendor-specific inference library for NVIDIA GPUs. We validate our compilation results by comparing them against Ansor's.

Benchmarks. Our evaluation benchmark uses four typical DNN models, including ResNet-50 [19] (CNN), LSTM [20] (RNN), NASNet [36] (a state-of-the-art CNN model obtained

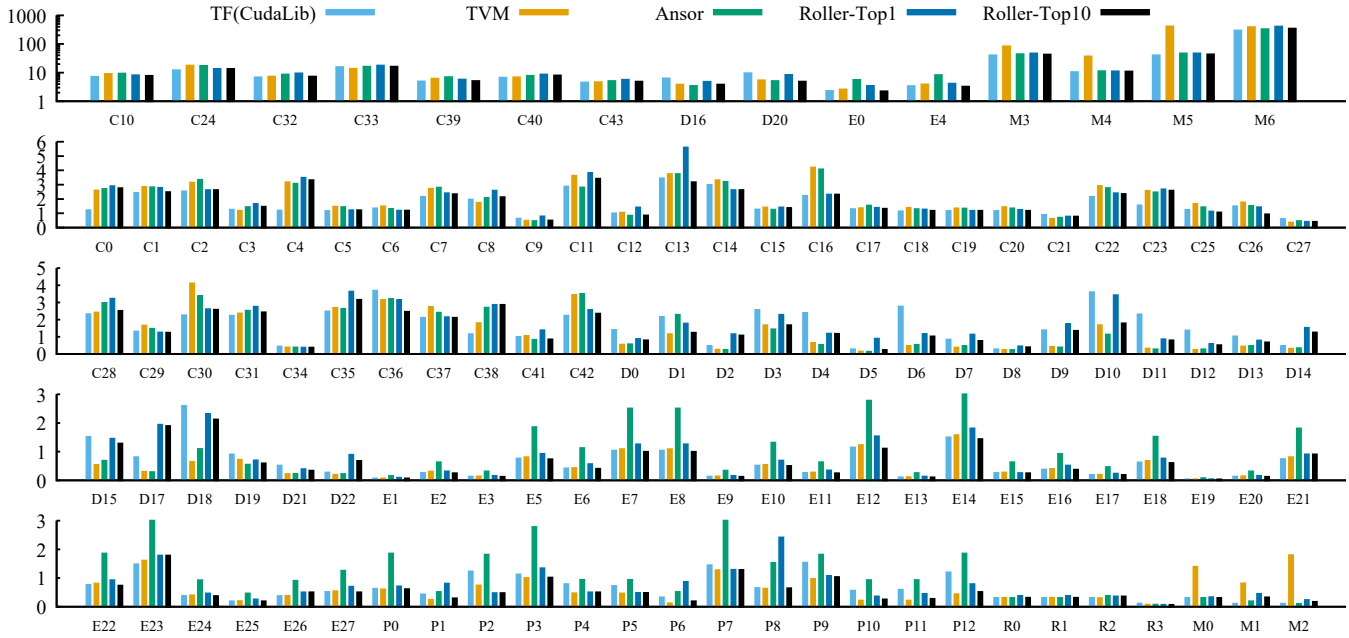


Figure 10: Operator performance on V100 GPUs (y-axis: average kernel execution time in ms).

by the neural architecture search), and BERT-Large [17] (transformer-based). We set the default batch size of each model to 128. From each model, we choose the most-frequently used operators to construct our operator benchmark. It contains 6 classes of operator type with total 119 operator instances with different configurations (7 MatMul operators, 44 Conv2D operators, 23 DepthwiseConv operators, 28 element-wise operators, 13 pooling operators, and 4 reduction operators). Table 1 lists a representative subset of operators as well as their configurations. The last column lists the corresponding abbreviation of each operator. The full list of the operator configurations is omitted due to page limit.

5.1 Evaluation on NVIDIA GPUs

This section first evaluates ROLLER's operator performance, compilation time, and scalability on large operators by comparing against the state-of-the-art tensor compilers and vendor libraries. We also evaluate the performance of ROLLER on TensorCore. Finally, we show the end-to-end model performance compared to existing DNN compilers and framework.

Operator performance. We first evaluate the performance of ROLLER generated kernels by comparing against TVM (i.e., AutoTVM with XGBoost tuning algorithm [16]), Ansor, cuBLAS (for matrix multiplication operators) and cuDNN (for convolution operators). Vendor libraries like cuBLAS and cuDNN are wrapped in TensorFlow to evaluate the performance. For the rest of operators (e.g., element-wise, reduce), we use TensorFlow's built-in kernel implementations. To amortize the overhead of data feeds/fetches in TensorFlow's session, we repeat the kernel running for 1,000 times in each session and calculate the average. We set the tuning steps for TVM and Ansor to 1,000 for each operator, same as Ansor's evaluation setup [33], and report the best results. We compare both the top-1 and the best from the top-10 kernels constructed by ROLLER, the latter can tolerate some hidden performance impacts from device compilers.

Figure 10 plots the average kernel performance for all the 119 operators in our benchmark, ordered by the operator type and ID. We plot the large operators (e.g., kernel time is larger than 5ms) in the top sub-figure in a log-scale for y-axis, and the other medium and small operators in the bottom 4 sub-figures [2]. First, compared to CUDA libraries (CudaLib), ROLLER could get comparable performance (i.e., within 10% performance) for 81.5% of the total operators, and can be even faster for 59.7% of them. We observe that the majority of operators that ROLLER performs worse are convolution operators with 3×3 or larger filters, which are usually implemented with a more efficient numerical algorithm (e.g., Winograd [23]) in cuDNN and hard to be expressed by the tensor expression. This is the reason Ansor and TVM are also slower than CudaLib in these cases. Second, compared to TVM and Ansor, ROLLER could also get comparable performance for 72.3% and 80.7% of the total operators respectively. The rest 27.7% and 19.3% of them are mainly small operators or with irregular tensor shapes, which are by natural hard to align with the hardware. However, these operators usually have relatively short kernel time, e.g., only 1.65ms and 1.16ms on average. Among 54.6% and 65.5% of the total

[2] Please find the complete results in our artifact.

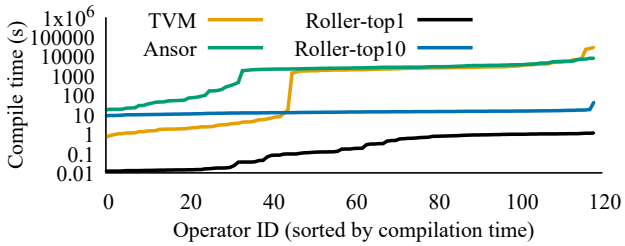


Figure 11: Compilation time for each operator.

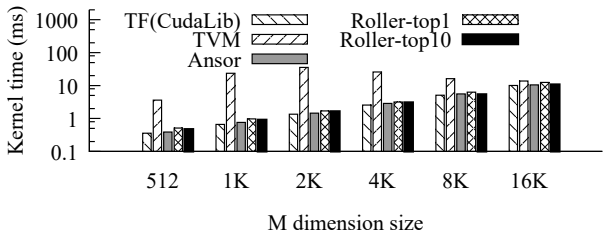


Figure 12: Kernel time for MatMul operator with different sizes of M in BERT-Large model, K=1024, N=4096.

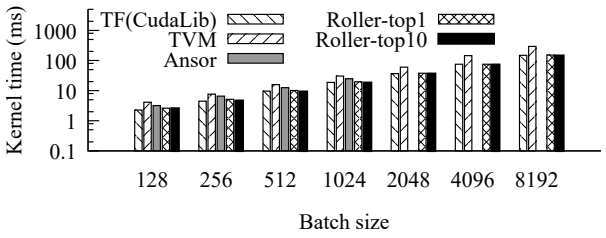


Figure 13: Kernel time for Conv2d operator with different batch sizes of N, where C=1024, H=14, F=2048, K=1, S=2.

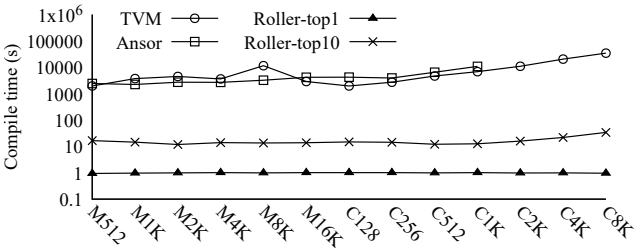


Figure 14: Compilation time for both MatMul and Conv2d operator with different batch sizes.

operators, ROLLER can even produce faster kernels than TVM and Ansor, respectively. We observe that the majority of these operators are large and time-consuming ones. As it shows in the top sub-figure where operators are larger than 5ms (up to 343ms), ROLLER could achieve better performance for most of these operators, e.g., by 1.85× and 1.27 × speedup over TVM and Ansor on average.

Compilation time. Given the comparable kernel performance, the major advantage of ROLLER is its fast compilation. Figure 11 compares ROLLER's compilation time against TVM and Ansor for all the operators. The operator ID is sorted by the compilation time for each line. The average operator compilation time for TVM is 0.65 hours and up to 7.89 hours. For the first 40 operators, which are mainly the element-wise, reduction, and pooling operators, TVM's compilation takes less than 10 seconds. This is because TVM's manually-written code templates for these operators can directly emit code without searching. However, Ansor generates search spaces for all the operators. Its compilation time takes 0.66 hours on average and up to 2.17 hours. In contrast, ROLLER's top-1 kernel results can be generated in 1 second for most operators and in 0.43s on average, which is *more than three orders of magnitude* faster. The major time is spent on the recursive constructing algorithm, which increases slightly with the growth of operator size, but quickly stabilizes as the recursive depth (to enlarge the *r*Tiles) is bounded by the limited memory capacity. To get the optimal kernels from the top-10 candidates, ROLLER's average compilation time is only 13.3 seconds. The major cost comes from the kernel code compilation with the device compiler and the evaluation on target devices.

Scale-out with operator size. We evaluate the scalability of ROLLER on larger operators by comparing with both CUDA libraries, TVM, and Ansor. We select a MatMul operator from the BERT model and a Conv2D operator from the ResNet mode, and scale them by setting different batch sizes. Figure 12 and Figure 13 show the performance comparisons. For the MatMul operator, both Ansor and ROLLER have a linear scalability over the batch sizes and comparable performance with CudaLib (i.e., cuBLAS). However, TVM's performance is relatively non-stable. For example, ROLLER can outperform TVM by average 11.2× and up to 36.1× for the batch size of 1024. For Conv2D operators, ROLLER can still achieve linear scalability over the batch size, and get slightly better performance than Ansor and TVM (by 1.25 and 1.54× on average). Note that Anosr is unable to search for a valid kernel for the batch size over 2048 using its default configurations. TVM can generate valid kernels, but the performance is scaled sublinearly for the larger batch sizes, e.g., ROLLER can achieve more than 1.9 × speedup for batch sizes greater than 2048.

Finally, Figure 14 compares the compilation time for the two operators with different batch sizes. The average compilation time of TVM and Ansor is 2.36 (up to 9.55) hours and 1.19 (up to 3.0) hours respectively. Moreover, their compilation time grows constantly with the growing of batch size. This is because that they are both based on ML-based search approach, whose search space usually increases exponentially with the operator size. In contrast, ROLLER produces the top-1 kernel in 1 second, and 16 seconds (up to 34 seconds) on average for the top-10 kernel.

Compile on TensorCore. ROLLER could easily support hardware tensor ISAs (e.g., TensorCore) by aligning the *r*Tile shape with the hardware instruction shape. We use the 16×16×16 WMMA instruction in ROLLER. We remove An-

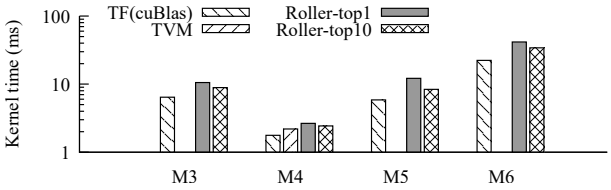


Figure 15: Matmul kernel time on TensorCore.

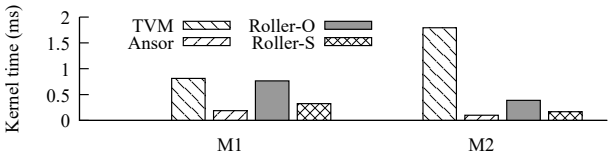


Figure 16: Performance for small operators.

sor in this experiment as it does not support TensorCore to our best knowledge. We select 4 large MatMul operators that are friendly to TensorCore in this experiment. Figure 15 shows the performance comparisons. As it shows, by constructing from the aligned *r*Tile shape, ROLLER can quickly produce good kernels on TensorCores, e.g., within a 43% performance gap to cuBLAS. Note that cuBLAS is highly optimized with a lot of hand-crafted optimizations on TensorCore. As a comparison, TVM fails to generate valid kernels for 3 of the 4 total operators with the default configurations. We try to increase the tuning steps from 1,000 to 10,000, it is still unable to find a legitimated kernel due to its poorly-defined search space.

Small operators and irregular tensor shape. ROLLER optimizes performance for small operators by shrinking the *r*Tile when there is insufficient parallelism. We demonstrate the performance of this optimization for the two small MatMul operators. Figure 16 compares the performance of the original *r*Tile configuration without sufficient parallelism (Roller-O), and the shrunken *r*Tile configuration (Roller-S) which matches the SM parallelism. As it shows, shrinking *r*Tile could significantly improve performance than the original kernel, e.g., by 2.3× on average. However, ROLLER is still slower than Ansor, e.g., by 50% on average, on small operators, even it is significantly faster than TVM by 6.6×. For such operators, we can further leverage search-based approach to fine-tune the configurations to obtain a better performance.

ROLLER compiles operators with irregular tensor shapes with two optimizations: i.e., *axis fusion* and *tensor padding* with bound parameter ε. We demonstrate their benefits on a representative set of irregular convolution operators, as shown in Figure 17. We compare the performance of ROLLER without any optimizations (Roller-B), with axis fusion (Roller-F), and further with tensor padding of ε from 0.4 to 1.0 (Roller-P0.4 and Roller-P1.0). All ROLLER's performances are the best one selected from the top-10 candidates. First, with axis fusion optimization, ROLLER is able to have more *r*Tiles that aligns with the tensor shapes, which improves the kernel performance by 1.5× on average. Moreover, with the tensor

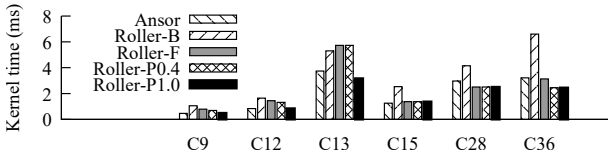


Figure 17: Performance for operators with irregular shapes.

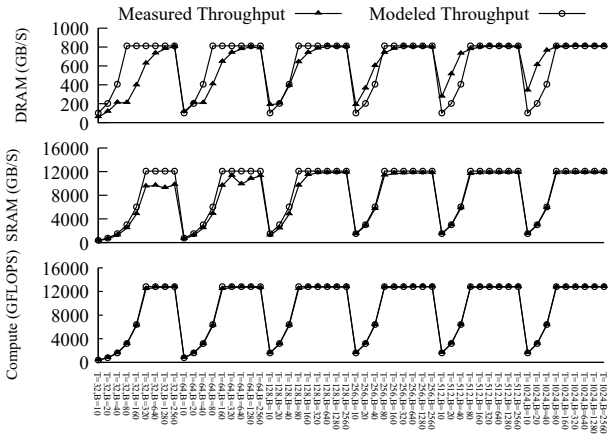


Figure 18: Memory throughput (DRAM and shared memory) and compute throughput from our micro-performance model and real measurement (X-axis: kernel configurations with different number of threads per block (T) and blocks (B)).

padding optimizations (e.g., at ε of 1.0), ROLLER can further improve performance than Roller-F by 1.4×. This is mainly because the number of legitimated kernels is very limited with smaller ε for irregular shapes. Increasing the ε allows ROLLER to have chance to select from more candidate kernels.

Micro-performance model. We conduct extensive experiments to validate the micro-performance model, including global memory throughput, shared memory throughput, and compute throughput, under different kernel configurations (i.e., different thread block and grid size). Figure 18 compares the performance estimated by our micro-performance model with that measured on real device. As shown, when the configuration is not aligned with the parallelism of execution units, i.e., thread number per block is less than 128 (4 warps), our model produces a relatively estimation error, especially for the DRAM throughput. This is also the case when there is insufficient parallelism (i.e., block number is less than 80). Thus, we can see our micro-performance model is accurate only for those shape-aligned configurations (i.e., *r*Tiles), as they fully exploit hardware efficiency. This also motivates us to choose only the aligned *r*Tiles, which greatly reduces the complexity of micro-performance model.

Kernel performance. We further study how close the performance of ROLLER generated kernels can approach the optimal. Since ROLLER's data pipeline model can naturally identify the bottleneck layer, e.g., DRAM, shared memory, or

Resource utilization	60-70%	70-80%	80-90%	90-100%
Operator #	6	13	22	78
Percentage	5%	11%	18%	66%

Table 2: The distribution of resource utilization at the saturated layer for different kernels.

Baseline	MemAlignn	EUAlign	ShapeAlign	BankAlign
1.0x	1.42x	1.88x	1.92x	1.94x

Table 3: Average accumulated performance improvement with different alignment optimization.

computation, we profile each generated kernel and compare the corresponding resource utilization at the saturated layer with the theoretical hardware limit. Table 2 lists the distribution of the resource utilization for the total 119 operators. The table shows most kernels saturate hardware resources, e.g., 66% of them utilize more than 90% of the theoretical limit. For the few under-utilized kernels, especially whose utilization is less than 80%, our investigation shows that they are mostly small operators with insufficient parallelisms.

To understand the impact of different alignment rules, we incrementally turn on each alignment optimization and evaluate its performance improvement. Table 3 shows the average speedup compared with the baseline (without any optimization). For example, EUAlign shows the kernels with the alignment on execution units and memory transaction alignment (MemAlign) can together improve the performance by 1.88x than the baseline. Bank alignment (BankAlign) has relatively small improvement because most kernels are already bank conflict free.

End-to-end model performance. We evaluate the end-to-end model performance of ROLLER by comparing against TensorFlow (TF), TensorFlow-XLA (TF-XLA), TensorRT (TF-TRT), and Ansor, which represent the state-of-the-art DNN framework, graph-level compiler, vendor-provided DNN engine, and DNN compiler with tensor compilation, respectively. Note that TensorRT is also the core engine in NVIDIA Triton inference server [8]. We omit TVM in this experiment as it usually requires an order of magnitude longer compilation time on tuning end-to-end models than Ansor [33]. ROLLER's end-to-end model compilation is implemented in Rammer (i.e., Rammer+Roller) by feeding the generated kernels into it. To create a fair baseline, we manually feed both the TVM and Ansor generated kernels for the same set of operators into Rammer, which are denoted as Rammer+TVM and Rammer+Ansor.

Table 4 lists the model execution time for each model compiled or executed by each compiler and framework. Note that TF-XLA fails to compile the BERT-Large and NASNet model (out-of-memory). TF-TRT also fails to run the BERT-Large model due to exceeding the maximum protobuf size limit (2GB) in its graph loading stage. For Ansor, we set the total tuning steps as 1,000 multiplied with the number of subgraphs for each model. However, Ansor also fails to produce

	BERT-Large	ResNet	NASNet	LSTM
TF	5,186	131	1,041	141
TF-XLA	OOM	112	OOM	98
TF-TRT	N/A	137	883	31
Ansor	46,847 (TVM)	122	927	84
Rammer+TVM	17,730	143	1,168	43
Rammer+Ansor	5466	137	1036	48
Rammer+Roller	4,850	142	1,005	20
Ansor compile-time	30.9h (TVM)	33.4 h	41.8h	11.3 h
Roller compile-time	371s	352s	668s	298s

Table 4: End-to-end model execution time (in milliseconds) and compilation time on V100 GPUs.

	TF(CudaLib)	TVM	Ansor
Better Performance	82.4%	65.5%	71.4%
Perf. within 5%	82.4%	67.2%	75.6%
Perf. within 10%	83.2%	73.1%	79.0%
Perf. within 50%	99.2%	93.3%	94.1%
Perf. within 90%	100.0%	100.0%	100.0%

Table 5: The percentage of better and comparable performant operators on NVIDIA K80 GPUs.

a legitimate program for BERT-Large models. Thus, for this case, we use TVM to compile the model. Note that, the performance of TVM for BERT-Large is about 2.6× slower than Rammer+TVM, as the default layout of the dense operator in TVM (i.e., NT) is different from that in Rammer (i.e., NN). First, for the ResNet and NASNet models, ROLLER can only achieve comparable and mostly slower performance than TF, TF-XLA, and TF-TRT (up to 26.7% slower compared to TF-XLA for ResNet). This major overhead in ROLLER is caused by the less efficient convolution kernels compared to cuDNN as explained before. However, for the BERT-Large and LSTM models, ROLLER can outperform all other frameworks and compilers, e.g., by 1.07× and 1.55× faster than the state-of-the-arts, i.e., TF for BERT-Large and TensorRT for LSTM. This mainly due to ROLLER's kernel construction favors large and regular operator shape, which are heavily used in the BERT-Large model. For both the BERT and LSTM models, since ROLLER can control to generate resource-efficient kernels by the scaling-up policy, it provides more opportunities for Rammer to co-schedule parallel kernels on the parallel SMs on GPUs. They together produce an efficient end-to-end program, which can even outperform TF-TRT by 1.55× for LSTM. Among all the implementations, Ansor can also produce very efficient programs for all the rest 3 models except for the BERT. However, it requires a long compilation time (29.3 hours on average). For the NASNet model, it reaches only 32% of the overall searching progress after tuning for 41.8 hours. In contrast, ROLLER only takes 422s on average to compile these models. This includes the graph-level optimization and the full-model compilation time in Rammer, which occupies about 41% of the total time on average.

Operator performance on K80 GPUs. We also evaluate ROLLER on the K80 GPUs. Table 5 shows the percentage of better or comparable performing operators (e.g., within 10%

	TF(RocLib)	TVM	Ansor
Better Performance	73.1%	58.8%	70.6%
Perf. within 5%	79.0%	62.2%	72.3%
Perf. within 10%	81.5%	62.2%	73.9%
Perf. within 50%	94.1%	84.0%	86.6%
Perf. within 90%	100%	100%	100%

Table 6: The percentage of better and comparable performant operators on AMD ROCm MI50 GPUs.

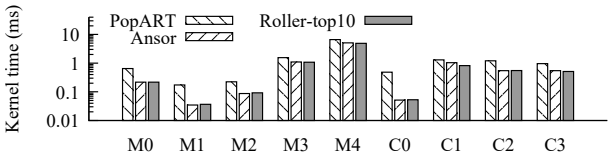


Figure 19: Operator performance on Graphcore IPU (y-axis in log-scale).

differences or 1.1× slow down) ROLLER generates for our operator benchmarks. Compared to CUDA libraries, TVM, and Ansor, ROLLER produces 82.4%, 65.5% and 71.4% better kernels for the whole operator benchmark. The percentage is relatively low for TVM mainly because the manual-crafted element-wise kernel templates in TVM are already highly-optimized. Finally, the average compilation time for all operators is 0.65 hours for TVM and 0.95 hours for Ansor respectively. In contrast, ROLLER's average compilation time is only 5.24 milliseconds for top-1 kernel and 12.3 seconds for top-10 kernel.

5.2 Evaluation on Other Accelerators.

Operator performance on AMD ROCm GPUs. We evaluate ROLLER on AMD ROCm GPUs by comparing it against ROCm libraries, TVM, and Ansor. Table 6 shows the percentage of operators that ROLLER can produce better or comparable performance (e.g., within 5% and 10% differences) in our operator benchmarks. Compared to the ROCm libraries (e.g., rocBlas), 73.1% of the total operators ROLLER can produce better kernels. This percentage is much higher than that on CUDA GPUs (59.7% and 54.6% for V100 and K80 GPUs). This is mainly because the libraries on CUDA GPUs are more mature than the ROCm GPUs, where ROLLER can help significantly. Compared to TVM and Ansor, ROLLER can also produce 58.8% and 70.6% better kernels. Similar to CUDA GPUs, the kernels that are slower by more than 10% are mostly small operator and those with irregular tensor shapes: the average execution time of these kernels are only 1.69ms and 1.57ms for TVM and Ansor, respectively. Finally, the average compilation time for all operators is 0.85 (up to 4.2) hours for TVM and 0.99 (up to 3.4) hours for Ansor, respectively. In contrast, ROLLER's average compilation time is 0.24 (up to 0.63) seconds for top-1 kernel and 7.69 (up to 49.0) seconds for top-10 kernel.

Operator performance on Graphcore IPU. We evaluate ROLLER on Graphcore IPUs. Due to the limited on-chip memory capacity, we only evaluate a set of small MatMul and Conv2D operators with different configurations. Figure 19 shows the average kernel time of each operator in log-scale, comparing against the Poplar-sdk library (i.e., PopART) provided by Graphcore and Ansor. Since TVM and Ansor do not have Graphcore backends, we use a modified version of Ansor in this experiment. As it shows, ROLLER can generate faster kernels than PopART for all operators, with an average of 3.1× and up to 9.2× speedup. Even comparing to Ansor, ROLLER can still construct comparable or even better kernels in most of operators, i.e., 2.9% average improvement. Note that Ansor still requires hours of tuning for each operator, as the device compiler on IPUs could take up to minutes to compile a program. However, ROLLER usually produce good kernels from the top-10 constructed candidates in several minutes. This time is mainly bottle-necked by the less-matured device compiler. It also brings more challenges to adopt the ML-based tensor compilers on these devices.

6 Discussion and Future Work

Optimization space compared with loop-based compiler. The abstraction of *r*Tile and data processing pipeline allows ROLLER to construct an optimization space overlapped with, but different from, existing DNN compilers (e.g., Ansor) [15, 33, 35]. As mentioned previously, these compilers view tensor compilation as nested loop optimizations. For example, Ansor allows only divisible tiling sizes along a tensor dimension to partition a loop axis evenly. This makes it usually perform worse for tensor shapes with prime dimensions. ROLLER instead focuses on maximizing hardware efficiency from the data-processing-pipeline view, allowing more aggressive optimizations, e.g., exploring non-divisible but hardware-aligned tiling sizes with fused adjacent axis and padded tensor shapes. Driven by our observation that most DNN operators are memory-bound, ROLLER fundamentally differs from existing DNN compilers by first optimizing data-tile throughput, i.e., maximizing reuse score rewards and aligning with hardware features, and then for parallelism. Such a trade-off inherently leads to fast compilation and good performance for operators with sufficient parallelism.

Optimization trade-off. ROLLER's design philosophy is based on an observation: large and dense operators tend to be major contributors to the execution time. This leads to a design trade-off: optimizing data reuse (i.e., maximizing pipeline throughput) as the primary optimization goal, and turning other hardware related optimizations into alignment constraints. Such trade-off results in fast compilation and high kernel quality for a majority of operators in mainstream workloads. For small operators, ROLLER further employs some adaptive mechanisms to trade-off among different optimiza-

tion goals, e.g., using a threshold to limit redundant work (§3.1) when there are insufficient results, employing an adapting *r*Tile shrinking process to increase parallelism (§3.2), etc.

Future work. ROLLER currently relies on high level device compiler, e.g., nvcc, to compile kernel code to executable. This sometimes introduces undesirable performance impacts and forces ROLLER to allocate registers conservatively. This is because the device compiler will implicitly allocate registers for intermediate values (e.g., loop variables). ROLLER cannot detect implicit register allocation beforehand, hence it is difficult to estimate and decide the precise register usage. One of our future work is to generate assembly (e.g., PTX for NVIDIA GPUs) code directly to avoid the side effects from the high level device compiler.

Moreover, although the key hardware information that affects performance, including memory bandwidth, capacity, and transaction length, is often available in the hardware specification, there are still some devices (e.g., mobile GPUs) lacking such information. Another future work is to leverage some profiling techniques [25] to disclose and quantify those hardware features.

ROLLER's HAL assumes hardware contains homogeneous computing units and symmetric memory accessing. However, we also observe that some devices have NUMA architecture. This makes it difficult for the micro-performance model to estimate *r*Tile performance, as the same tile will perform differently at different locality under NUMA architecture. We leave this issue as future work.

Finally, the optimization for sparse kernel may also violate the assumption of homogeneous workload in a DNN kernel and make the micro-performance model inaccurate. Some tiles with a larger degree of sparsity may perform differently from dense tiles. ROLLER assumes a higher level, sparsity-aware compiler (e.g., SparTA [34]) will address this issue.

7 Related Work

Most tensor compilers treat DNN operators as nested multi-level loop computation, which essentially defines a large space with a combinatorial complexity. TVM [15] inherits the insight from Halide [27] and describes DNN operators as loop optimization schedule primitives. Later, AutoTVM [16] extends TVM to apply an ML-method to search for the best configurations from manually written code templates. FlexTensor [35] proposes to automatically explore the space without manual templates. Ansor [33] further advances such automation. It generates an even larger search space considering a hierarchical code structure and adopts an evolution algorithm to find performant kernels. Compilers like Tiramisu [14], AKG [32], and Tensor Comprehensions [29] apply polyhedral-based techniques to loop optimization, which transforms the loop into an integer programming problem and finds a good configuration with a solver. All these approaches rely on a huge search space to provide good kernel, which leads to long compilation/solving time. ROLLER explores a different approach to construct *r*Tiles that align with hardware features.

Tensor Processing Primitives (TPPs) [18] define a set of 2D-tensor operators to compose complex operators on high-dimensional tensors, providing limited expressiveness. In contrast, ROLLER does not limit the dimension of tile shape and can be applied to general tensor expressions. The OpenAI Triton [28] is a programming framework and compiler for developing block-based GPU kernels. Triton relies on programmers to decide the block size and block scheduling, while this is the key problem ROLLER addressed by considering both hardware features and tensor shapes. MLIR [5] and Tensor IR [10] plan to support block-level (i.e., tile) computation representation in their IRs. ROLLER's *r*Tile abstraction and the *r*Program construction are compatible with these initiatives.

Graph-level DNN compilers like XLA [11], TVM [15], and Rammer [26] focus on cross-operator optimizations, e.g., operator fusion/co-scheduling. ROLLER's kernel generation is compatible with these compilers. ROLLER's *r*Tile abstraction complements the rTask concept in Rammer [26] as it provides an efficient way to construct an rTask.

Finally, some works focus on operator-specific optimizations. CUTLASS [7] is a template for implementing matrix-multiplication. An analytical model [24] is proposed to find the best loop-level optimization configuration only for convolution operators on multi-core CPUs. And DREW [30] proposes a new way to optimize Winograd convolution using data compression [31]. ROLLER's optimization approach is general for DNN operators on various devices.

8 Conclusion

ROLLER takes an unconventional approach to deep learning compiler. Instead of relying on costly machine learning algorithms to find a good solution in a large search space, ROLLER generates efficient kernels using a recursive construction-based algorithm that leverages the new *r*Tile abstraction with much fewer shapes that align with multiple hardware features. The constructed program can be evaluated by a micro performance model, without running on a real device every time. As a result, ROLLER can compile high-performance kernels in seconds, even in less mature accelerators. More importantly, ROLLER offers a unique and significantly more efficient approach for new AI hardware vendors to build competent vendor-specific DNN libraries, bridging the ecosystem gap to market leaders and thereby facilitating innovations in AI accelerators.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof.Yufei Ding, for their extensive suggestions.

References

[1] AMD ROCm Platform. https://github.com/RadeonOpenCompute/ROCm.

[2] CUDA Basic Linear Algebra Subroutine library. https://docs.nvidia.com/cuda/cublas/index.html.

[3] CUDA NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[4] IPU PROGRAMMER'S GUIDE. https://www.graphcore.ai/docs/ipu-programmers-guide.

[5] MLIR. https://mlir.llvm.org/.

[6] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[7] NVIDIA cutlass. https://github.com/NVIDIA/cutlass.

[8] NVIDIA TRITON INFERENCE SERVER. https://developer.nvidia.com/nvidia-triton-inference-server.

[9] ONNX. https://onnx.ai/.

[10] TensorIR. https://discuss.tvm.apache.org/t/rfc-tensorir-a-schedulable-ir-for-tvm/7872.

[11] XLA. https://www.tensorflow.org/xla.

[12] AMD Radeon Instinct™ MI50 Accelerator, accessed 2018 Nov. https://www.amd.com/en/products/professional-graphics/instinct-mi50.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. USENIX Association.

[14] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[16] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3389–3400. Curran Associates, Inc., 2018.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[18] Evangelos Georganas, Dhiraj D. Kalamkar, Sasikanth Avancha, Menachem Adelman, Cristina Anderson, Alexander Breuer, Narendra Chaudhary, Abhisek Kundu, Vasimuddin Md, Sanchit Misra, Ramanarayan Mohanty, Hans Pabst, Barukh Ziv, and Alexander Heinecke. Tensor processing primitives: A programming abstraction for efficiency and portability in deep learning workloads. *CoRR*, abs/2104.05755, 2021.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[21] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[22] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.

[23] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[24] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. Analytical characterization

and design space exploration for optimization of cnns. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 928–942, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Rendong Liang, Ting Cao, Jicheng Wen, Manni Wang, Yang Wang, Jianhua Zou, and Yunxin Liu. Romou: Rapidly generate high-performance tensor kernels for mobile gpus. In *The 28th Annual International Conference On Mobile Computing And Networking (MobiCom 2022)*. ACM, February 2022.

[26] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.

[27] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[28] Philippe Tillet, H. T. Kung, and David Cox. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*, page 10–19. Association for Computing Machinery, New York, NY, USA, 2019.

[29] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[30] Ruofan Wu, Feng Zhang, Jiawei Guan, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. Drew: Efficient winograd cnn inference with deep reuse. In *Proceedings of the ACM Web Conference 2022*, WWW '22, page 1807–1816, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. Poclib: A high-performance framework for enabling near orthogonal processing on compression. *IEEE Transactions on Parallel and Distributed Systems*, 33(2):459–475, 2022.

[32] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. Akg: Automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

[34] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. Deep-learning model sparsity via tensor-with-sparsity-attribute. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, 2022.

[35] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. pages 859–873, 03 2020.

[36] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

Walle: An End-to-End, General-Purpose, and Large-Scale Production System for Device-Cloud Collaborative Machine Learning

Chengfei Lv
Zhejiang University & Alibaba Group

Chaoyue Niu*
Shanghai Jiao Tong University & Alibaba Group

Renjie Gu, Xiaotang Jiang, Zhaode Wang, Bin Liu, Ziqi Wu, Qiulin Yao, Congyu Huang, Panos Huang, Tao Huang, Hui Shu, Jinde Song, Bin Zou, Peng Lan, Guohuan Xu
Alibaba Group

Fei Wu
Zhejiang University

Shaojie Tang
University of Texas at Dallas

Fan Wu, Guihai Chen
Shanghai Jiao Tong University

Abstract

To break the bottlenecks of mainstream cloud-based machine learning (ML) paradigm, we adopt device-cloud collaborative ML and build the first end-to-end and general-purpose system, called Walle, as the foundation. Walle consists of a deployment platform, distributing ML tasks to billion-scale devices in time; a data pipeline, efficiently preparing task input; and a compute container, providing a cross-platform and high-performance execution environment, while facilitating daily task iteration. Specifically, the compute container is based on Mobile Neural Network (MNN), a tensor compute engine along with the data processing and model execution libraries, which are exposed through a refined Python thread-level virtual machine (VM) to support diverse ML tasks and concurrent task execution. The core of MNN is the novel mechanisms of operator decomposition and semi-auto search, sharply reducing the workload in manually optimizing hundreds of operators for tens of hardware backends and further quickly identifying the best backend with runtime optimization for a computation graph. The data pipeline introduces an on-device stream processing framework to enable processing user behavior data at source. The deployment platform releases ML tasks with an efficient push-then-pull method and supports multi-granularity deployment policies. We evaluate Walle in practical e-commerce application scenarios to demonstrate its effectiveness, efficiency, and scalability. Extensive micro-benchmarks also highlight the superior performance of MNN and the Python thread-level VM. Walle has been in large-scale production use in Alibaba, while MNN has been open source with a broad impact in the community.

1 Introduction

To provide intelligent services for millions or even billions of smartphone users in industry, the mainstream paradigm lets mobile devices send requests with raw data and lets the cloud return results after data processing and model execution. However, this paradigm encounters three major bottlenecks: (1) *High Latency:* The network latency between each mobile device and the cloud plus the request processing latency of the cloud is in seconds, which is unacceptable for some real-time interactive applications. For example, the practical latency requirements of computer vision (CV), natural language processing (NLP), and recommendation tasks are in hundreds or even tens of milliseconds; (2) *High Cost and Heavy Load:* On the device side, uploading raw data will incur high cellular data usage, if Wi-Fi is not available. On the cloud side, receiving and storing enormous amounts of raw data from a massive number of mobile devices, processing data with diverse and sophisticated ML algorithms, and returning results in time, inevitably cause high overhead. For example, the size of 60s-long video or audio is in tens of MB, and the size of raw data for recommendation per user per day is in MB. Further multiplied by the scale of mobile devices, the total size of raw data is huge; and (3) *Data Security and Privacy:* Uploading the raw data with sensitive contents (e.g., personal information and user behaviors) may raise serious security and privacy concerns of users. Storing and processing raw data on the cloud may suffer from the risk of data breach.

By deconstructing the cloud-based ML paradigm, we can find that it simply regards mobile devices as interactive interfaces, but ignores the fact that mobile devices after 10 years of development can now undertake an appropriate load of data processing and model execution. Therefore, it does not leverage the natural device-side advantages of being close to users and data sources, thereby reducing latency and communication cost, mitigating the cloud-side load, and keeping private data on local devices. To overcome the bottlenecks of the mainstream cloud-based ML paradigm, the device-cloud collaborative ML paradigm emerged, which advocates offloading part of ML tasks to mobile devices and letting the cloud and the mobile devices collaboratively accomplish the ML tasks. Existing work tends to focus on the algorithmic decisions (e.g., device-cloud task splitting strategy [29] and collaboration/interaction paradigm [34]) in either inference or training phase and normally for a specific application (e.g., video analytics [6,11,31], text processing [5], recommendation [17,44]).

*Chaoyue Niu is the corresponding author (rvince@sjtu.edu.cn).

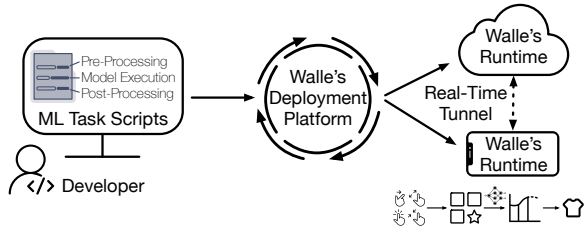


Figure 1: Walle from the perspective of an ML task developer.

However, practical industrial scenarios tend to involve the full cycle of diverse CV, NLP, and recommendation applications to serve millions or even billions of mobile devices, building a general-purpose system to put device-cloud collaborative ML in large-scale production becomes a new requirement.

We build an end-to-end system, called Walle, the overall goal of which is to support general device-cloud collaboration (e.g., single device-cloud and multiple devices-cloud) in each phase of diverse ML tasks through exchanging any necessary content (e.g., data, feature, sample, model, model update, and intermediate result). As shown in Figure 1, Walle supports the whole cycle of ML tasks (i.e., pre-processing, model training and model inference, and post-processing) on both mobile devices and cloud servers in both development (e.g., the practical need of frequent experimentation and deployment for daily ML task iteration) and runtime (i.e., ML task execution and device-cloud data transfer) stages. By following the philosophy of building a general-purpose system rather than integrating massive application-specific or platform-specific solutions, Walle functions as a fundamental ML infrastructure with standard APIs and keeps the light-weight limit of mobile APPs, having supported 1,000+ kinds of CV, NLP, and recommendation tasks in large-scale production.

During building Walle, we encounter several practical requirements and challenges that motivate our design decisions. Walle is oriented by ML tasks and consists of a deployment platform, a data pipeline, and a compute container, catering to ML task deployment, input preparation, and execution, respectively. (1) For the compute container, one major requirement is to decouple ML task iteration from the monthly/weekly update of mobile APPs, making the classical method of integrating new functionalities into APPs infeasible. Another key requirement is to support diverse ML tasks with high performance across different operating systems (OS) and heterogeneous hardware of mobile devices and cloud servers. This requires to build a tensor compute engine in C/C++ and do operator-level and computation graph-level optimizations for each hardware backend. Two dominant strategies are manual optimization (e.g., in almost all ML engines), the workload of which is quite heavy that only some common cases can be covered; and auto tuning (e.g., in TVM [9]), which cannot support runtime optimization and is infeasible in industrial scenarios that involve massive heterogeneous devices or require frequent/quick ML task iteration. Based on the tensor compute engine, the libraries should be implemented to cover pre-processing, model training and inference, and post-processing as well as mobile devices and cloud servers in a unified way, rather than in a separate and incomplete way, like NumPy, OpenCV, TensorFlow (Lite), and PyTorch (Mobile). Without integrated design, the high performance of the tensor compute engine cannot be exposed to different libraries, the workload of optimizing each library on heterogeneous backends is heavy, and the package is large; (2) for the data pipeline, the overarching goal is to prepare raw data, which can come from different sources and are structured in various formats, as device-side or cloud-side ML model input. The mainstream paradigm of uploading all the device-side raw data to the cloud for aggregate processing is inefficient and error-prone; and (3) for the deployment platform, its key requirement is to manage, release, and deploy ML tasks for numerous mobile devices in a fine-grained, timely, and robust way, given massive ML task deployment requirements, intermittent device availability, and potential task failure.

We overcome the key challenges above and build Walle. (1) We choose dynamically-typed, widely-used Python as the script language for developing ML tasks in Walle and implement a Python VM as the core of the compute container by refining CPython in two aspects: one is to abandon the global interpreter lock (GIL) and support task-level multi-threading with VM isolation and data isolation; and the other is to perform tailoring for practical device-side need. Such design enables daily ML task iteration. At the bottom of the compute container, we implement a tensor compute engine along with standard data processing and model execution libraries, called MNN [2]. MNN first introduces a novel geometric computing mechanism to decompose the transform and composite operators into atomic operators, thereby dramatically reducing the workload of manually optimizing hundreds of operators for tens of backends; and then introduces a novel semi-auto search mechanism to quickly identify the best backend with runtime optimization for a series of operators. At the top of the compute container, we expose MNN to Python thread-level VM as standard APIs, supporting the whole cycle of diverse ML tasks with standard data input. (2) For the data pipeline in Walle, we mainly build a new on-device stream processing framework to enable processing user behavior data at source. The key novelty is managing the trigger conditions of multiple stream processing tasks to generate different features with a trie structure for concurrent triggering. We also establish a real-time tunnel to transfer device-side fresh features to the cloud for use. (3) Regarding the deployment platform of Walle, we propose to manage task entity with git, categorize task-related files into shared and exclusive ones to facilitate multi-granularity deployments, and release tasks with an efficient push-then-pull method and in steps.

Walle is now part of Alibaba's ML backbone infrastructure, being invoked more than 153 billion times per day and supporting more than 0.3 billion daily active users, 30+ mobile APPs, and 300+ kinds of ML tasks. MNN is open source now

with 6,600+ stars and 1,300+ forks on GitHub, and also is in production use in 10+ other companies. Evaluation of Walle in example real applications (i.e., livestreaming and recommendation) and platform statistics demonstrate effectiveness, efficiency, and scalability. Micro-benchmarks of MNN and Python thread-level VM show superiority.

We summarize the key contributions as follows: (1) Walle is the first end-to-end, general-purpose, and large-scale production system for device-cloud collaborative ML, masking hardware and software heterogeneity at the bottom, and supporting diverse ML tasks with daily iteration cycle and high performance at the top; (2) the compute container in Walle comprises MNN, which introduces geometric computing to sharply reduce the workload of manual operator-level optimization, and semi-auto search to identify the best backend with runtime optimization; and a Python VM, which is the first to abandon GIL and support task-level multi-threading, and also is the first to be ported to mobile devices; (3) the data pipeline in Walle introduces on-device stream processing with trie-based concurrent task triggering to enable processing user behavior data at source; and (4) the deployment platform in Walle supports fine-grained task release and deployment to billion-scale devices with strong timeliness and robustness.

2 Preliminaries

In this section, we first expound the background and motivation of building a general-purpose system for device-cloud collaborative ML. We then elaborate on the major design challenges. We finally draw the system requirements.

2.1 Background and Motivation

Machine Learning Task. From a developer's perspective, an ML task comprises scripts (e.g., codes in Python), resources (e.g., data, models, and dependent libraries), and configurations (e.g., trigger conditions mainly for specifying where and when to trigger the ML task). The whole workflow of an ML task can be divided into three phases or sub-tasks[1]: pre-processing, model execution, and post-processing. In the pre-processing phase, raw data from multiple sources are cleaned, integrated, and processed to extract features and generate samples, which are then fed into models. In the model execution phase, a model is loaded to do training or inference. In the post-processing phase, the results of model inference are processed (e.g., by applying some ranking policies or business rules) to finally serve users.

Motivating Industrial Applications for Walle. In Alibaba, there are now at least hundreds of online ML tasks to serve billion-scale daily active users with mobile devices in tens of business scenario, where CV, NLP, and recommendation tasks roughly account for 30%, 10%, and 60% of the total tasks and run billion, one hundred billion, and billion times every day, respectively. In particular, (1) typical CV-type application scenarios include livestreaming, visual image search, short video analysis, augmented reality, and security checkup, where the major tasks include key frame detection, image segmentation and classification, item recognition, facial recognition and effects, human keypoint and pose detection, and porn detection; (2) typical NLP-type application scenarios include livestreaming and voice navigation, where the major tasks include automatic speech recognition, text to speech, text analysis, and text generation; and (3) typical recommendation-type application scenarios include item re-ranking, intelligent refresh, message popping, and page rearrangement, where the key tasks include click-through-rate prediction, click-conversion-rate prediction, user state recognition, and user intent detection.

Need for Device-Cloud Collaborative System. The applications raise strict latency requirements on ML tasks. In general, (1) CV tasks need to process each image in 30ms; (2) NLP tasks require to process a 5s-long audio segment in 500ms or process an audio stream with latency less than 100ms; and (3) recommendation tasks need to generate outputs in 300ms. In addition, the raw data from massive users input to ML tasks are huge. For example, (1) for CV tasks, the size of a 60s-long, 1080p, and 8Mbps video is roughly 60MB; (2) for NLP tasks, the size of a 60s-long WAV/PCM audio is around 10MB; and (3) for recommendation tasks, one user normally generates thousands of pieces of raw data per day, each piece in the size of KB. Furthermore, raw user data are more or less sensitive, raising security and privacy concerns.

The practical requirements above make the mainstream cloud-based ML paradigm infeasible and motivate us to adopt device-cloud collaborative ML. The key principle is that an ML task can be executed not only on the cloud but also on mobile devices, rather than purely on the cloud. During the execution of an ML task, mobile devices can work as a relay of the cloud, and vice versa. The choice of which side to execute which phase is flexible and should incorporate the practical need of the ML task and the characteristics of the cloud and mobile devices. For example, choosing which side for pre-processing should consider whether the side is near data source. Further observing the industrial need to support diverse ML tasks and massive devices, we are motivated to build an end-to-end and general-purpose system that can put device-cloud collaborative ML in large-scale production.

2.2 Practical Challenges

A device-cloud collaborative ML system faces some practical challenges that span the execution, input preparation, and deployment stages of ML tasks as follows.

Execution Challenges. (1) *Long Iteration Cycle:* The common update cycle of a mobile APP includes development, testing, and integration of new functionalities (e.g., ML tasks to be deployed in our context), as well as APP store review and release to massive mobile devices in batches. As a result, most APPs is updated weekly, while some super APPs (e.g., Mobile

[1] We call ML sub-tasks also as ML tasks for convenience.

Taobao, a shopping APP owned by Alibaba with 0.3 billion daily active users) are updated monthly. However, ML tasks require frequent experiments/deployments in nature, such that the effectiveness of different ML algorithms and models can be quickly verified, and the optimal feature combination and hyper-parameters can be identified; (2) *Heterogeneous Backends:* The cloud servers and mobile devices significantly differ in hardware (e.g., CPU, GPU, NPU, instruction set architecture (ISA), and memory) and OS (e.g., Android, iOS, Windows, and Linux). Among mobile devices, the ecosystem is even more fragmented; (3) *Diverse ML Tasks:* Industrial applications involve many kinds of ML tasks, requiring diverse model structures (e.g., convolutional neural network (CNN), recurrent neural network (RNN), transformer, generative adversarial network (GAN), and deep interest network (DIN)). Meanwhile, pre-processing and post-processing also involve lots of image, text, and numerical processing methods; and (4) *Limited Device Resources:* Each mobile APP has only one process. For Mobile Taobao, the maximum RAM is only 200MB, and the package size cannot exceed 300MB.

Input Preparation Challenges. (1) *Atypical User Behavior Data:* For CV and NLP tasks, most raw data (e.g., image, video, text, and audio) are in standard formats, the pre-processing of which can be supported by standard libraries. Another major data source, the pre-processing of which cannot be directly supported, is each user's diverse behaviors in time and page series during interacting with a mobile APP and is essential to many ML (especially, recommendation) tasks. Conventionally, all the users' behavior data are uploaded to the cloud, far away from source, for stream processing with Flink. To enable pre-processing at source, there, however, does not exist an on-device stream processing framework; (2) *Diverse Trigger Conditions:* ML tasks tend to need many features. Each feature corresponds to a stream processing task and its trigger condition. How to efficiently manage multiple trigger conditions for concurrent task triggering is non-trivial.

Deployment Challenges. (1) *Massive Task Deployment Requirements:* In Alibaba, the size of active ML tasks is at least in hundreds, and the mobile devices to be covered can reach the scale of billion. The release of each ML task also needs to incorporate APP versions, device-side and user-side differentiation; (2) *Intermittent Device Availability:* Mobile devices are with unstable wireless connections and allow only one APP to run on the foreground, while users tend to switch APPs frequently. Therefore, from the perspective of a certain APP, each device's availability is dynamic. Conventional push (e.g., based on persistent connection) or pull (e.g., based on polling) deployment method cannot guarantee timeliness and incurs high load on the cloud; (3) *Potential Task Failure:* A mobile APP runs as a single process. The failure of any task will lead to the crash of the whole APP, seriously impacting user experience. Further, due to the massive task deployment requirements, it is impractical to test each pre-release task on all relevant types of real devices.

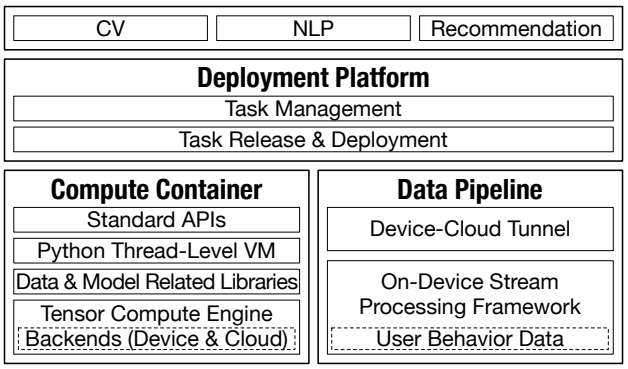


Figure 2: Architecture of Walle.

2.3 System Requirements

Given the challenges above, the design of an device-cloud collaborative ML system should meet some requirements.

The ML task execution environment needs to satisfy: (1) *Quick Task iteration:* ML tasks can be iterated daily on a mobile APP, reliving the dependence on the APP's original update cycle; (2) *Cross Platform:* OS-level and hardware-level heterogeneity should be masked; (3) *High Performance*: Optimization need to be specific to heterogeneous hardware backends of mobile devices and cloud servers; (4) *Universality:* Diverse CV, NLP, and recommendation tasks should be supported. The pre-processing, model execution, and post-processing phases of each ML task should be supported in an end-to-end way; and (5) *Light Weight:* The whole package size needs to be small, especially for mobile devices.

The ML task input preparation pipeline needs to first introduce *a new on-device stream processing framework* with concurrent task triggering ability to enable processing user behavior data at source. To enable the cloud to consume the generated features (e.g., for feature fusion or model inference) far away from source with low latency, a real-time tunnel between mobile devices and the cloud also needs to be built.

The ML task deployment platform should guarantee: (1) *Multi-Granularity:* Task release needs to support uniform, device-level grouping, user-level grouping, or even extremely device-specific policy; (2) *Timeliness:* A large number of mobile devices can be covered in short time; and (3) *Robustness:* Task deployment must put stability in the first place.

3 Walle: Architecture and Design Rationale

Guided by the system requirements, we build Walle. We first introduce the whole architecture and then design rationale.

3.1 Architecture Overview

As shown in Figure 2, the compute container in Walle comprises: (1) a cross-platform and high-performance tensor compute engine at the bottom; (2) data processing and model execution libraries based on the tensor compute engine; (3) a

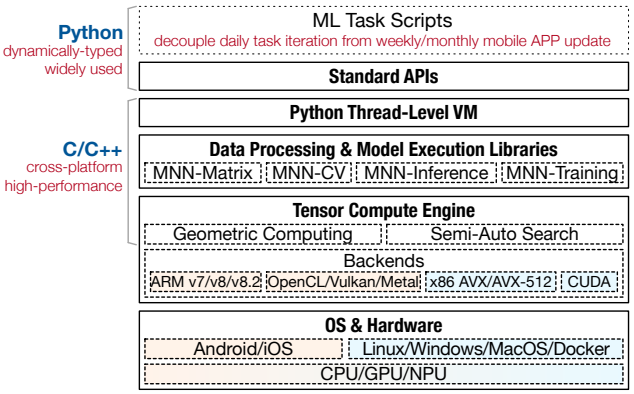


Figure 3: Architecture of compute container.

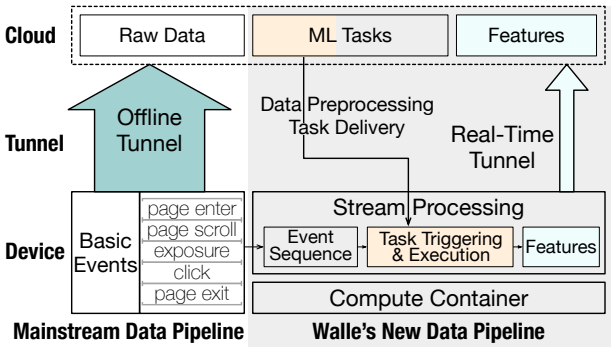


Figure 4: Architecture of data pipeline.

Python thread-level VM; and (4) standard APIs at the top. The data pipeline introduces: (1) an on-device stream processing framework; and (2) a real-time device-cloud tunnel. The deployment platform in Walle comprises: (1) a task management module; and (2) a task release and deployment module.

3.2 Design Rationale

Rationale of Compute Container. As shown in Figure 3, on the top, we choose Python as the script language, because Python is widely used in developing ML algorithms and also is a dynamically-typed and interpreted language. To support executing the Python scripts of ML tasks on different platforms, especially on resource-constraint mobile devices, we implement a Python VM by refining CPython and perform tailoring for the practical need of a mobile APP. Further considering the characteristics of ML task execution, including concurrent triggering of many tasks, independence across different tasks, and sequential execution of different phases in each individual ML task, we abandon GIL in Python VM and support task-level multi-threading by first binding each ML task with a thread and then conducting thread isolation. Such Python VM-based design endows the compute container with the capability of dynamic task delivery, decoupling daily ML task iteration from monthly/weekly mobile APP update.

At the bottom, we implement a tensor compute engine in C/C++ for cross-platform and high-performance considerations. The cores are the novel mechanisms of geometric computing and semi-auto search, as shown in Figure 5. In particular, geometric computing extracts a new atomic operator from transform operators, by leveraging the nature of coordinate transformation as well as the linear mapping between an element's coordinate and its memory address. As a result, all the transform and composite operators, accounting for roughly 49% of all the operators, can be decomposed to the atomic operators, reducing 46% of the workload of manually implementing and optimizing 124 operators for 16 kinds of backends from algorithm, ISA, memory, and assembly. Then, to quickly identify the backend available on a mobile device or a cloud server to execute a computation graph with a series of operators at the minimum cost, semi-auto search is applied in runtime to find the optimal implementation algorithm with the optimal parameters for each operator on each available backend. The parameter search is converted to solving a constrained optimization problem, by incorporating the hardware properties of the backend and the sizes of the implementation algorithm's inputs. Based on the tensor compute engine, we implement the libraries of scientific computing, image processing, model inference, and model training, and expose them to Python VM as standard APIs, supporting the whole cycle of diverse ML tasks with standard data input.

Rationale of Data Pipeline. The architecture is depicted in Figure 4. First, a user's behaviors are naturally recorded as a time-level event sequence, based on which the page-level event sequence can be created by aggregating the events in the same pages. Then, the trigger condition of a stream processing task can be specified by a sequence of event/page ids. To support concurrent triggering, we model matching multiple trigger conditions with the event sequence as a string matching problem with multiple wildcard patterns and propose to organize trigger conditions with a trie, such that if a new event comes, all the triggered tasks can be picked out for execution. Given a stream processing task can be triggered frequently over the continuously generated event sequence, while the size of one-time output is small, we design a collective storage mechanism to reduce the frequency of write. Finally, to upload the output of on-device stream processing with low latency, we leverage persistent connection to implement a real-time tunnel, transferring up to 30KB data within 500ms.

Rationale of Deployment Platform. We first manage the task entity with git and categorize task-related files into shared and exclusive ones, according to how many devices can use the files in common. The file categorization further facilitates the uniform and customized policies of task deployment. To guarantee the timeliness of task deployment, we propose a novel push-then-pull method based on transient connection, where the push functionality reuses the existing client-side http request for business services, while the pull functionality is via content delivery network (CDN) and Alibaba cloud enterprise network (CEN). For the robustness of task deployment, we introduce task simulation test with the cloud-side

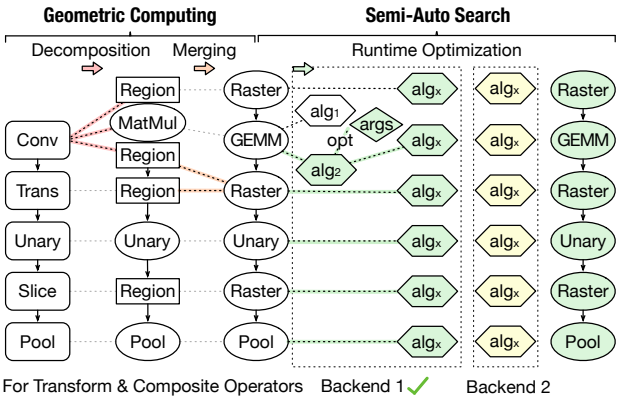


Figure 5: Geometric computing and semi-auto search.

compute container before release and enforce releasing task in steps, while allowing rollback in the case of task failure.

In what follows, we present the design and implementation details of the compute container in Section 4, the data pipeline in Section 5, and the deployment platform in Section 6.

4 Compute Container in Walle

We introduce the compute container in a bottom-up way: MNN, a tensor compute engine along with the data processing and model execution libraries; Python thread-level VM; and standard APIs of MNN.

4.1 Tensor Compute Engine

Tensor computation can be viewed as the basis of data processing and ML, and the operators of underlying tensor computation can be divided into four categories: (1) *Atomic Operators*, which function as the basic unit of backend optimization, such as some common unary operators (e.g., taking square) and binary operators (e.g., addition, subtraction, multiplication, and division); (2) *Transform Operators*, which change the shape and/or reorder the elements, such as transpose, slicing, concatenation, and permutation; (3) *Composite Operators*, which can be decomposed into the atomic and transform operators, such as 3D convolution and pooling, normalization, exponential linear unit, and long short-term memory cell; and (4) *Control-Flow Operators*, including if and while.

Geometric Computing. Currently, MNN can support $N_{aop} = 61$ atomic operators, $N_{top} = 45$ transform operators, $N_{cop} = 16$ composite operators, and $N_{fop} = 2$ control-flow operators. The workload of implementing and optimizing the operators for all $N_{ba} = 16$ backends in MNN is $O((N_{aop} + N_{top} + N_{cop}) \times N_{ba} + N_{fop} = 1954)$. Further considering the workload involving the atomic and control-flow operators is unavoidable, we turn to reducing the workload involving the transform and composite operators, which roughly accounts for half of the whole load and will grow in the future (e.g., as more composite operators are required to support more kinds of deep neural network (DNN)). Our key idea is to extract a new atomic operator, called "raster", from the transform operators. Then, both the transform operators and the composite operators can be decomposed into the raster operator and the atomic operators. Since only the atomic and raster operators need to be optimized for each backend, the whole workload becomes $O((N_{aop}+1) \times N_{ba} + N_{top} + N_{cop} + N_{fop} = 1055)$, reducing roughly 46% of the workload. Now, the problems become what is the raster operator and how to implement it. We propose a geometric computing mechanism as follows.

In essence, the basic functionality of the transform operators is to move an element from a memory address to another memory address, or from geometry, is to transform the coordinate of the element to another coordinate. In addition, the memory address is a deterministic linear function of the coordinate. Moreover, given a certain transform operator, the formula of coordinate transformation can be determined. As a result, with the coordinate of an element in the input or output tensor, typically the element's index in the input or output tensor, the original memory address and the memory address after movement can also be determined. The raster operator is introduced to move the elements between the input and output tensors according to the memory addresses and by traversing the coordinates. We take slicing for example. A is a 2×4 matrix, placed in contiguous memory addresses with a unique identifier/pointer. The slicing of A by leaving only the second row is denoted as B, which is a 1×4 matrix. For an element $B_{i,j}$ with the row index i and the column index j (i.e., the coordinate (i, j)) in B, its memory identifier relative to B's unique identifier is $i \times 4 + j$, which is linear with the coordinate, where the coefficients $(4, 1)$ are called the strides. According to the definition/rule of slicing (i.e., $B_{i,j} = A_{i+1,j}$), the coordinate of the corresponding element $A_{i+1,j}$ in A is $(i+1, j)$, and the relative memory identifier is $(i+1) \times 4 + j = 4i + j + 4$, where the coefficients $(4, 1)$ are the strides, and the intercept 4 is called offset. The raster operator can realize the functionality of slicing by iterating the coordinates $\{(i, j) | 0 \le i < 1, 0 \le j < 4, i, j \in \mathbb{Z}\}$ and moving each $A_{i+1,j}$ to $B_{i,j}$ using their memory addresses.

In practical implementation of the raster operator, we introduce a supporting concept, called "region", which contains an input tensor, the range of coordinate, as well as the linear mappings between an element's coordinate and its memory addresses in the input and output tensors, which are called "views" and can be specified by the strides and offsets. In addition, after operator decomposition, some raster operations can be merged for optimization. One policy is called vertical merging, which mainly deals with two successive raster operations, skips indirect references, and operates on the original tensor; and the other policy is called horizontal merging, which handles two parallel raster operations with the same region and keeps only one raster operation.

Atomic Operator Optimization. Specific to the atomic operators, including the raster operator, we incorporate hardware heterogeneity and optimize the implementation from the

perspectives of algorithm, ISA, memory, and assembly. (1) The algorithm-level optimization is specific to some compute-intensive operators, typically convolution and matrix multiplication. We take more efficient algorithms, including Winograd and Strassen algorithms, to sharply reduce the number of multiplications; (2) the ISA-level optimization leverages single instruction multiple data (SIMD), such as ARM Neon and x86 AVX512, for speedup. To adequately exploit data-level parallelism in SIMD, we carefully design data layout and data packing. Specifically, we take a new NC/4HW4 layout [35] and a channel-major packing for convolution; (3) the memory-level optimization focuses mainly on reducing the number of read and write as well as improving the contiguity of memory allocation. In particular, for matrix multiplication, we apply tiling and memory reordering; and (4) the assembly-based optimization can achieve instruction-level speedup. We implement core operators with hand-written assembly codes and carefully apply some optimizations, such as loop unrolling, software pipelining, and instruction reordering.

Semi-Auto Search. Data processing and model execution normally involve a series of operators (i.e., the atomic, raster, and control-flow operators after decomposition). Meanwhile, different backends have different implementations and optimizations for the operators, and a mobile device or a cloud server tends to have several backends available. The global goal of semi-auto search is to identify the backend with the minimum cost. The cost of each backend is the sum of all the operators with the optimal implementations. To identify the optimal implementation algorithm for a certain operator on a certain backend, the optimal parameters of each possible algorithm need to be found. This is converted to a constrained optimization problem that can be quickly solved, where the objective is computation or memory cost, and the constraints incorporate the hardware constraints of the backend and the sizes of the algorithm's inputs. We formulate the whole process of semi-auto search and introduce the details as follows.

We let BA denote the set of all available backends and let $op_1 \rightarrow op_2 \rightarrow \ldots \rightarrow op_n$ denote the series of n operators for execution. The cost of a backend $ba \in BA$ is defined as

$$C_{ba} = \sum_{i=1}^{n} C_{op_i,ba}, \tag{1}$$

where $C_{op_i,ba}$ denotes the cost of the operator op_i with the optimal implementation on the backend ba. The goal of semi-auto search is to find the backend with the minimum cost, which can be expressed as

$$\arg\min_{ba \in BA} C_{ba}. \tag{2}$$

Then, the problem is how to compute each $C_{op_i,ba}$. For each operator op_i and the backend ba, we let $algs(op_i, ba)$ denote all feasible implementation algorithms with the optimal parameters. Then, $C_{op_i,ba}$ is defined as

$$C_{op_i,ba} = \min_{alg \in algs(op_i,ba)} \frac{Q_{alg}}{P_{ba}} + S_{alg,ba}, \tag{3}$$

where (1) Q_{alg} denotes the number of elementary calculations in the algorithm alg, which can be obtained given the ("optimal" here) parameters and the sizes of the inputs; (2) P_{ba} represents the performance of the backend Ba. In MNN, for a CPU-type backend, if the backend ba supports ARMv8.2-FP16, P_{ba} empirically takes 16 times the frequency; otherwise, P_{ba} takes 8 times the frequency. For a GPU-type backend, P_{ba} is empirically set to the number of floating point operations per second (FLOPS) by manual testing; and (3) $S_{alg,ba}$ denotes the scheduling cost of the algorithm alg on the backend ba. In MNN, for a CPU-type backend, $S_{alg,ba}$ is set to 0; and for a GPU-type backend, $S_{alg,ba}$ is empirically set and mainly considers the time of data transfer. Now, the remaining problem is for an operator op_i, a backend ba, an implement algorithm alg, and the sizes of the inputs, how to determine the optimal parameters of the algorithm. In practice, we formulate it into a constrained optimization problem, where the objective is to minimize the computation or memory cost, and the constraints mainly include the width of SIMD unit, the number of registers, the number of threads, and the sizes of the inputs. In addition, we focus mainly on optimizing the following parameters: the packing size in SIMD, the tile size in matrix multiplication, the block unit in the Winograd algorithm, and the reduction of the elementary calculations using the Strassen algorithm. We take the optimization of the title size in matrix multiplication for example. We let A denote an $a \times e$ matrix, let B denote an $e \times b$ matrix, let t_e denote the tile size along the axis with the equal size, let t_b denote the tile size along the axis of B's columns, and let N_r denote the number of registers. The optimization objective is minimizing the times of memory read and write. The formula of the optimization problem is given as follows:

$$\begin{aligned} \min_{t_e,t_b} \quad & \frac{e}{t_e} \times \frac{b}{t_b} \times (a \times t_e + a \times t_b + t_e \times t_b), \\ \text{s.t.} \quad & t_e \times t_b + t_e + t_b \leq N_r, \end{aligned} \tag{4}$$

which can be solved efficiently in runtime.

Compared with manual search, which optimizes the implementation algorithms with some common parameters for each operator case by case, semi-auto search not only can sharply reduce the workload but also can find the optimal parameters with higher probabilities. Regarding why not adopt auto tuning in TVM, it does not exploit manual experience in operator optimization, consumes long time of static compilation due to the large search space at the operator and graph levels for a certain backend, and cannot support runtime optimization. Most importantly, given the restriction on executable files and just-in-time (JIT) compilation on iOS devices for security [4], the compiled models generated by TVM must be linked into mobile APPs with monthly/weekly update and cannot be daily iterated as desired. Therefore, TVM is infeasible in industrial applications that involve a large number of heterogeneous devices or require frequent/quick task iteration (e.g., updating the deployed ML models). In contrast, our design of the tensor

compute engine essentially leverages manual operator-level optimization for heterogeneous backends to narrow down the space of semi-auto search, thereby supporting deploying models as regular resource files and further facilitating runtime optimization and daily ML task iteration in Python VM. Another benefit is that the package size of mobile APPs will not increase in the long term for more and more ML tasks.

4.2 Data and Model Related Libraries

With the tensor compute engine, we implement the libraries of scientific computing and image processing for the pre-processing and post-processing phases of an ML task, as well as the libraries of model inference and model training. In particular, the scientific computing and image processing libraries can be regarded as the optimized implementations of NumPy [21] and OpenCV [28] in terms of light weight and high performance. The light weight means that the sizes of libraries can be reduced without manual tailoring. The original sizes of NumPy 1.9.3 and OpenCV 3.4.3 are 2.1MB and 1.2MB, and decrease to 51KB and 129KB in MNN, respectively. The high performance indicates that the performance optimization of the underlying tensor compute engine can be inherited to the libraries, avoiding the extra workload. We introduce the implementations of the libraries as follows.

Scientific Computing & Image Processing. We use the atomic, raster, and control-flow operators to support array creation and manipulation routines, binary operations, linear algebra, logic functions, padding arrays, random sampling, mathematical functions, etc, in the scientific computing library; and to support image filtering, geometric and miscellaneous image transformations, drawing functions, color space conversions, etc, in the image processing library.

Model Inference & Model Training. We currently provide two modes of model inference in MNN, called session and module. The module mode can support the control-flow operators, which are required by transformer, dynamic RNN, etc, whereas the session mode cannot. The session-based model inference can be divided into four steps: (1) load a model, create a session, arrange all the operators in the computation graph according to the topological ordering, and apply for the tensors that all the operators need; (2) given the shape of each input tensor and the definition of each operator, compute the shapes of all the tensors; (3) perform geometric computing, particularly, first decompose the transform and composite operators into the atomic and raster operators, and then do vertical and horizontal merging for raster operators; and (4) identify the optimal backend with semi-auto search, request memory for each operator and execute in sequence, and return the inference result. In the second step, the control-flow operators require the intermediate result to determine the following execution order and thus cannot be supported in the session mode. To solve this problem, when loading the model in the first step, the module mode splits the computation graph into modules (i.e., sub-graphs) iteratively, according to the positions of the control-flow operators. Then, the execution of each module is the same as that of the session.

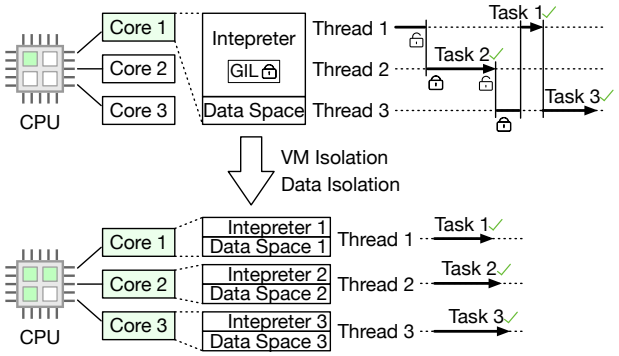


Figure 6: Python thread-level virtual machine.

We implement model training by adding two common optimizers: stochastic gradient descent (SGD) and adaptive moment estimation (ADAM). At the bottom, we add the gradient operators of all the atomic operators and one raster operator.

4.3 Python Thread-Level Virtual Machine

Most ML tasks are implemented in Python and require a Python VM to execute the Python scripts. We choose the official and the most widely-used Python compiler and interpreter, called CPython [43]. However, there exist two key problems in the porting process of CPython, especially for resource-constraint mobile devices. The first problem is that the size of the package is large. For example, CPython 2.7.15 contains 500+ scripts in C and 1,600+ libraries, including many redundant functionalities for mobile APPs. The second problem is that CPython cannot support multi-threading to improve efficiency. CPython originally cannot support concurrent programming and introduces GIL for multi-processing. GIL allows only one thread to be processed at one time within a process. However, each mobile APP has only one process and does not allow multi-processing. How to support task-level multi-threading in Python VM becomes a problem.

To reduce the package size, we tailor the functionalities, libraries, and modules for the practical need of Mobile Taobao. (1) *Functionality Tailoring:* CPython first compiles Python code into bytecode with the file suffix ".pyc" and then interprets the bytecode for execution. By leaving the compile phase on the cloud and sending only the bytecode to mobile devices for execution, we can delete all the compile modules, saving 17 scripts in C. (2) *Library and Module Tailoring:* We keep 36 necessary libraries (e.g., abc, type, re, and functools) and 32 modules (e.g., zipimport, sys, exceptions, and gc). After package tailoring, we implement a light-weight Python interpreter for mobile devices, which is the first in industry. For example, on ARM64-based iOS, the package size decreases from 10MB+ to only 1.3MB.

Regarding multi-threading, we abandon GIL and further design and implement the first Python thread-level interpreter in

industry, supporting the concurrent execution of many tasks. As shown in Figure 6, each task is scheduled to a certain thread, which creates an independent VM and contains the VM runtime and task-related data. For thread safety, the key is to perform thread-level VM isolation and data isolation, which pin a VM to its thread and further pin the context of VM runtime to the thread. (1) *VM Isolation:* The lifecycle of the original Python VM is pinned to the process, each process having one VM. We need to modify the creation of VM instances such that a process can hold multiple thread-level VMs, each VM having its independent lifecycle. In CPython, VM is defined as a struct in C, called PyInterpreterState. When CPython starts, one PyInterpreterState instance will be initialized. We modify the initialization of CPython, particularly creating and initializing a PyInterpreterState instance for each thread. (2) *Data Isolation:* Besides VM itself, the context of VM runtime (e.g., type system, module, and task-related data) should also be isolated on the level of thread, avoiding the concurrency problem of multi-threading without the protection of GIL. We adopt the thread-specific data (TSD) technique for data isolation, such that each thread has its own data space, and different threads cannot access the same data simultaneously. We mainly apply TSD to type system, buffer pool, object allocation, and garbage collection.

4.4 Standard APIs

We expose the cross-platform libraries of data processing and model execution through Python VM to support ML tasks. For pre-processing and post-processing, the scientific computing and image processing APIs are consistent with the original APIs of NumPy and OpenCV to be developer-friendly, such as matmul, swapaxes, concatenate, split, resize, warpAffine, warpPerspective, cvtColor, GaussianBlur, etc. For model inference and model training, the APIs of common model-level and data-level operations are exposed, such as data loading, model loading and saving, session creation and execution, optimizers, hyper-parameter setting, loss computing, etc.

5 Data Pipeline in Walle

We detail the on-device stream processing framework and the real-time device-cloud tunnel in the data pipeline.

5.1 On-Device Stream Processing Framework

The key design goal is to support stateful computation over unbounded data stream on single device. A user's behavior data in a mobile APP tracked with accurate timestamps form stream. The processing of user behavior data is stateful, where the intermediate results are buffered in memory or stored locally for later usage. The resources of single device are limited, which implies that the trigger conditions of many stream processing tasks should be well managed. We introduce on-device stream processing from event sequence

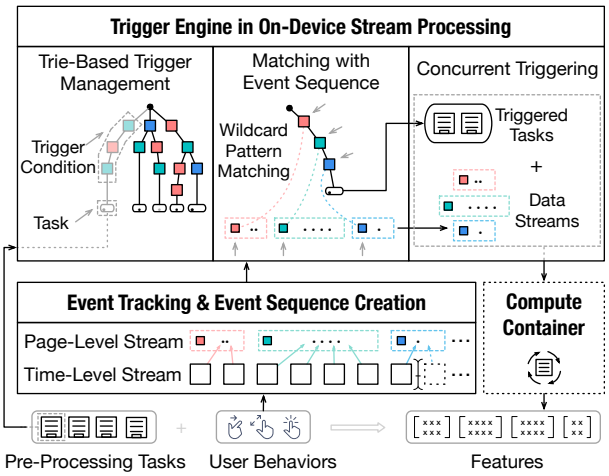


Figure 7: Workflow of on-device stream processing.

creation, trigger management, task triggering, task execution, and collective storage. The workflow is depicted in Figure 7.

Event Sequence Creation. When a user interacts with a mobile APP, the user's behaviors will be tracked as events. There are five major kinds of basic events: page enter, page scroll, exposure, click, and page exit. Each kind of event is recorded with a unique event id, a page id, a timestamp, and event contents (e.g., the item id for exposure-type event and the graphical widget id for click-type event). Since a user's behaviors are naturally in time series, the time-level event sequence can be directly created. To further benefit processing the events within a certain page or cross pages, the page-level event sequence is created by aggregating the events between the enter and exit events of the same pages.

Trigger Management. A stream processing task over the event sequence contains scripts and configurations, where the scripts implement the data processing algorithm, and the configurations mainly include a trigger condition. In particular, the trigger condition can be specified by a sequence of trigger ids, where a trigger id can be an event id or a page id.

For a certain mobile device, it needs to efficiently maintain multiple pre-processing tasks to generate different features for diverse ML tasks, such that as an event comes, all relevant tasks can be triggered immediately. The key is to organize trigger conditions for quick matching. The trivial method of storing trigger conditions in a list is inefficient, because of the need to traverse the entire list each time. In fact, the matching of multiple trigger id sequences with the event sequence (with both event and page ids) can be modeled as a string matching problem with multiple wildcard patterns. Therefore, we leverage the data structure of prefix tree, called a trie, for efficient trigger management. More specifically, the trie has three kinds of nodes: start, middle, and end nodes. The trie's root is the unique start node. A trigger id is a middle node. An end node, which stores the stream processing tasks, is a leaf node of the trie, and vice versa. When a new stream processing task comes, the trigger id sequence will be extracted as a sequence

of middle nodes, and a pair of start and end nodes will be added to the first and the last places of the node sequence, respectively. Then, the depth-first search is performed over the current trie from the root. If a path is completely matched with the node sequence, then the stream processing task will be added to the leaf node; otherwise, the mismatched nodes will be added to the trie as a new sub-tree, the root of which is the last matched node in the depth-first search process. We note that each path of the trie corresponds to a unique trigger condition, and the leaf node stores the stream processing tasks with the same trigger condition. If two trigger id sequences have common prefixes, then they will be put in the same sub-tree, and the middle nodes in the path from the trie's root to the sub-tree's root correspond to the common trigger ids.

Task Triggering. When a new event (with an event id and a page id) comes, the set of triggered tasks will be returned. First, two lists of trie nodes are introduced to record the concurrent matching states of multiple trigger conditions and to avoid being blocked by wildcard pattern matching. The static pending list stores all the children of the trie's root, which correspond to the first trigger ids in all the trigger conditions and always keep active for matching. The dynamic pending list stores the desired next nodes of the trigger conditions in the ongoing matching. For an event in the stream, if its event/page id matches the trigger id of any node in the static or dynamic list, then each child of the node will be checked for whether it is an end node. If the child is an end node, then the stream processing tasks in the end node will be returned; otherwise, the child, as a new desired next node, will be added to a buffer of the dynamic list. At the end of task triggering for the event, the dynamic list will be replaced by the buffer, and the buffer will be refreshed.

Task Execution. When a task is triggered, the scripts will be run in the compute container to process relevant events. Besides standard data processing and mode execution APIs of the compute container, to facilitate the extraction of relevant events from the event sequence and the processing of event contents, the stream processing framework also provides some basic functions as follows: (1) *KeyBy*, which returns the events matched with a given key; (2) *TimeWindow*, which returns the events in a given time window; (3) *Filter*, which returns the events filtered by a defined rule; and (4) *Map*, which processes the event contents with a defined function.

Collective Storage. For each stream processing task, its outputs, typically features, are saved as a table using SQLite. Considering the fact that a stream processing task can be triggered for several times, while the size of one-time output is small, a collective data storage API is encapsulated over SQLite to reduce the number of write, thereby improving performance. In particular, a buffering table will be created in memory, and the output of a stream processing task is first written to the buffering table. If the number of write reaches a certain threshold or a read operation is invoked, the buffering table will be written into the database once.

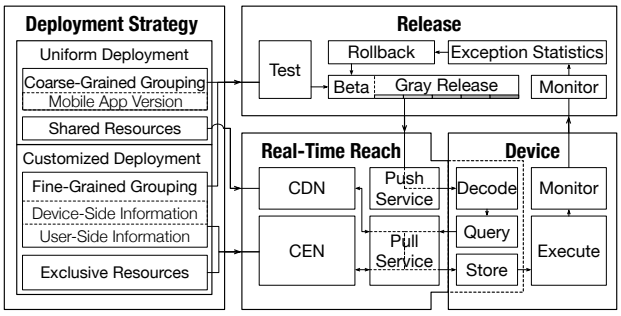


Figure 8: Workflow of deployment platform.

5.2 Real-Time Device-Cloud Tunnel

Besides for local use, the output of on-device stream processing can also be uploaded to the cloud for real-time use. We implement a device-cloud tunnel based on the persistent connection. The secure sockets layer (SSL) protocol is optimized to reduce the time of connection establishment, encryption, and decryption. The data are compressed before transfer and are decompressed after transfer. To deal with high throughput, a fully asynchronous service framework is built on the cloud.

6 Deployment Platform

We introduce the details of ML task management, release, and deployment. The whole workflow is shown in Figure 8.

Task Management. Git [41] is adopted to achieve the isolation of different tasks and the version control of a certain task, while supporting collaborative development with access control. In particular, the entire task management is regarded as a git group; each business scenario corresponds to a git repo (repository); each task in a business scenario corresponds to a branch; and each version of a task corresponds to a tag.

Besides the management of task entity, the task-related files, especially the resources (e.g., data and model) which can be large in size, are also managed in a fine-grained way to support uniform and customized deployments. The files are divided into two categories: one is the shared files, which can be used by a large number of mobile devices (e.g., the devices with a certain version of APP); and the other is the exclusive files, which can be used only by a small number of devices or even a specific device. The shared and exclusive files can be requested efficiently via CDN and CEN, respectively.

Task Release & Deployment. A uniform or customized policy can be taken to deploy tasks on targeted devices. The uniform policy supports task release grouped by the APP version, while the customized policy can further support grouping by device-side information (e.g., OS and its version or device performance) and user-side information (e.g., age or habit). According to the number of devices in a group, the coarse-grained uniform deployment normally involves only shared files, whereas the fine-grained customized task deployment not only can involve shared files but also can involve exclusive files. In the extremely personalized scenarios, the

customized policy supports deploying a certain kind of task but with user-specific/exclusive files to each individual device.

Regarding task release, we take a novel push-then-pull method. We reuse the existing client-side business request to implement push, by adding a mobile device's local task profile into the http header and letting the cloud compare it with the latest task profile. If a new task needs to be released and takes the uniform deployment policy, then the cloud responds with the CDN address of the shared task files. If the new task takes the customized deployment policy, the cloud first determines which group the mobile device belongs to by rule matching and then responds with the CDN address of the shared files or the CEN address of the exclusive files. After receiving the response from the cloud, the mobile device can pull the task files using the CDN or CEN address from the nearest node. Considering the fact that the client-side business request is frequent, while the speeds of CDN and CEN are fast in practice, the timeliness of task deployment can be guaranteed.

To guarantee the stability of task release and deployment, the simulators of the mobile APP with different versions for different OS can be created with the compute container on the cloud for testing a pre-release task extensively. Upon passing the simulation testing, a beta release is conducted to deploy the task only on a few targeted devices. After passing the beta release, the gray release is forced to be performed in steps, covering all the targeted devices incrementally. The deployment platform is also equipped with an exception handling module, which can monitor the failure rate of the task in real time and also can rollback immediately if the failure rate exceeds a certain threshold.

7 Evaluation of Walle

We evaluate Walle in two major application scenarios of Alibaba. We also extensively conduct benchmark testing for MNN, Python thread-level VM, and real-time tunnel. We finally report the statistics of the deployment platform.

7.1 Performance in E-Commerce Scenarios

Compute Container in Livestreaming. E-commerce livestreaming has brought a brand new form of online shopping to billion-scale users. In 2020, the gross merchandise value (GMV) of livestreaming in Mobile Taobao exceeded 400 billion RMB. One key ML task in this scenario is highlight recognition, which is to locate the time points of a streamer in introducing attractive information about items.

Under the conventional cloud-based ML paradigm, a video stream is uploaded from each streamer's mobile device to the cloud for highlight recognition, which mainly includes the detection and recognition of items as well as the facial detection and the voice detection of streamers. Due to the large number of online streamers, the long length of their video streams, and the stringent latency requirement of highlight recognition, the load of the cloud is so heavy that only part of video streams and only a few sampled frames can be analyzed, which becomes a key bottleneck in practice.

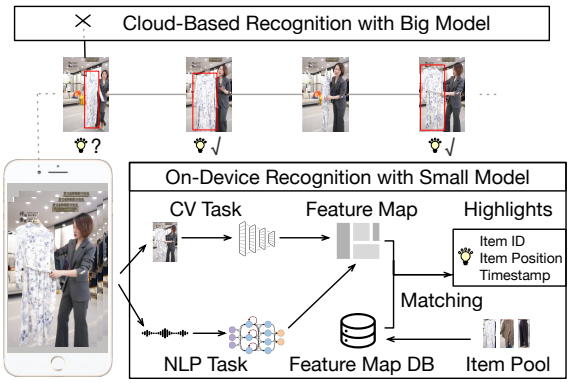


Figure 9: Workflow of device-cloud collaborative highlight recognition in e-commerce livestreaming.

Table 1: Model information and inference latency in device-side highlight recognition.

	Item Detection	Item Recognition	Facial Detection	Voice Detection
Model	FCOS [40]	MobileNet [25]	MobileNet [25]	RNN
Parameter Size	8.15M	10.87M	2.06M	8K
Huawei P50 Pro	56.92ms	25.68ms	41.42ms	0.07ms
iPhone 11	33.71ms	29.74ms	22.58ms	0.01ms

With Walle, we can offload the highlight recognition task with light-weight models to a streamer's mobile device and implement a device-cloud collaborative workflow, as shown in Figure 9. If the device-side models can recognize the highlights in a video stream with high confidences, then these highlights can be directly shown to users after post-processing. Only those highlights, which are recognized with low confidences on the mobile device and account for roughly 12% in practice, need to be processed by cloud-side large models. After passing cloud-side recognition, the rate of which is around 15%, the highlights will be delivered to the mobile device.

Through device-cloud collaboration, the numbers of streamers and video streams covered with highlight recognition dramatically increase, while the load of the cloud is also sharply relived. In particular, business statistics show that compared with the cloud-based paradigm, the new device-cloud collaborative workflow increases the number of streamers with highlight recognition by 123%; reduces the computing load of the cloud per highlight recognition by 87%; and increases the size of daily recognized highlights per unit of cloud cost by 74%. We also evaluate the performance of the compute container in Walle, when supporting highlight recognition on Huawei P50 Pro and iPhone 11. The total latency is 130.97ms and 90.42ms, respectively. In particular, the network architectures, the parameter sizes, and the inference latency of the adopted models are listed in Table 1. The results above demonstrate the high performance of our compute container and the practical effectiveness of device-cloud collaboration.

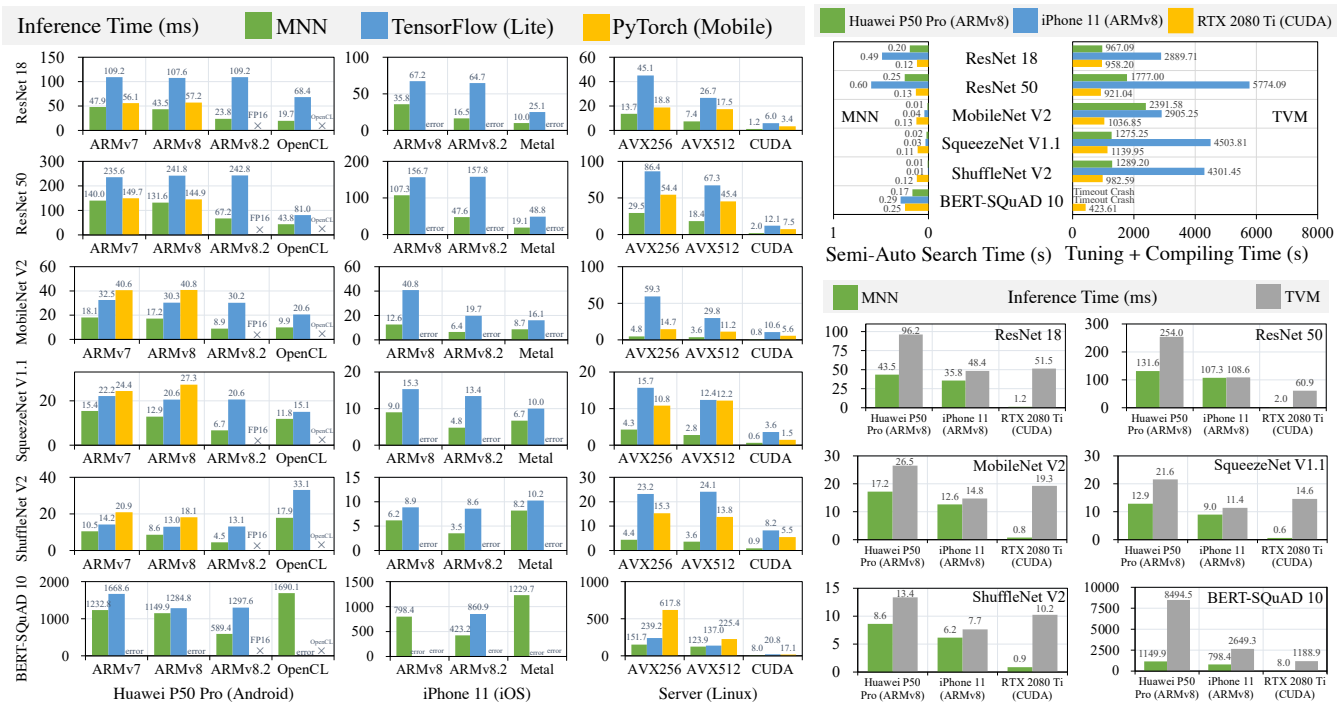


Figure 10: MNN vs. TensorFlow (Lite), PyTorch (Mobile), and TVM on different backends of mobile devices and cloud servers.

Data Pipeline in Recommendation. In Alibaba's cloud-side and device-side recommendation models, item page-view (IPV) feature, which records a user's behaviors (e.g., add-favorite, add-cart, and purchase) in the detailed page of an item, is of significant importance. To generate IPV feature, under the conventional cloud-based paradigm, all the users' raw event data are uploaded to the cloud for stream processing and mixed with user ids for explicit identification. The time-level event sequence from each mobile device is split into multiple homogeneous sequences, one sequence containing a certain kind of event. To obtain the IPV feature of each individual user, the cloud performs join operations with user id and page id as keys over all the users' events, which is quite time-consuming, resource-consuming, and error-prone.

With the on-device stream processing framework in the data pipeline, each mobile device needs to process only a small size of the corresponding user's local events, which is more efficient and more natural. In fact, the IPV feature invokes the generation process of the page-level event sequence. The input is the time-level event sequence. The trigger condition is the page exit event. The triggered stream processing task is to aggregate all the events (i.e., to cluster the same kind of events and gather statistics between the enter event and the exit event of the page). Since the raw contents in each event contain redundant fields (e.g., device status), a filtering is applied to the event contents. Further considering the fact that the IPV feature is first encoded (e.g., through RNN) in recommendation models, by using the model inference API of the compute container, the encoding process can also be offloaded to mobile devices.

We first show the size reductions from raw event data, to IPV feature, and to IPV encoding. On average, one IPV feature is around 1.3KB in size, involving 19.3 raw events in the size of 21.2KB, and one IPV encoding is only 128 bytes. This indicates that compared with the conventional paradigm of transferring raw event data to the cloud for stream processing, our new IPV data pipeline can save more than 90% of communication cost. Besides communication efficiency, we also compare the latency of on-device and cloud-based stream processing. By analyzing over 10,000 practical cases (randomly sampled from the case pool of 2 million online mobile clients) of processing raw events into IPV features, the average on-device latency is only 44.16ms. In contrast, using Alibaba's internal version of Flink, called Blink, the average latency of producing one IPV feature is 33.73s. In particular, the cloud-based stream processing is over 2 million online users' raw events and consumes 253.25 compute units (CU), where 1 CU denotes 1 CPU Core plus 4GB memory; the error rate of IPV feature generation is 0.7%; and the average latency is analyzed over 10,000 randomly sampled normal cases. These results reveal that compared the mainstream cloud-based data pipeline, Walle's new data pipeline can indeed reduce device-cloud communication cost and cloud-side load, while improving the timeliness and validity of feature.

7.2 Benchmark Testing

We first compare MNN with TensorFlow (Lite) and PyTorch (Mobile) on Android and iOS devices as well as Linux servers. For device-side testing, we use Huawei P50 Pro and iPhone 11, covering the backends of ARMv7, ARMv8, and ARMv8.2

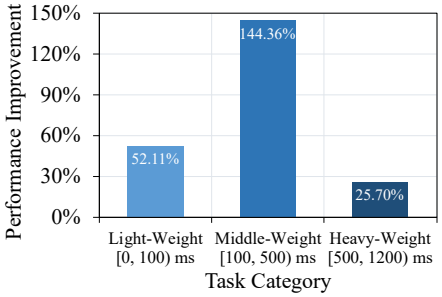


Figure 11: Python thread-level VM vs. CPython (30 million ML task executions).

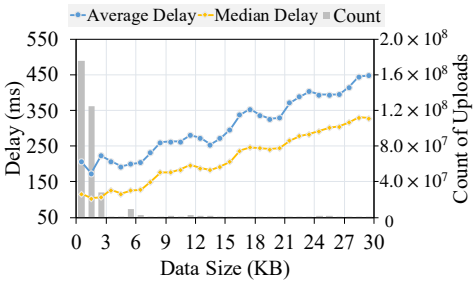


Figure 12: Delay of real-time tunnel (analyzed over 364 million uploads).

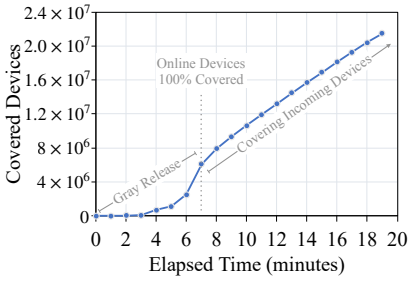


Figure 13: Timeliness of ML task deployment (22 million devices).

with single thread as well as OpenCL and Metal. For server-side testing, we use AMD Ryzen 9 3900X (x86), Alibaba Cloud's ecs.g6e.4xlarge (Intel Xeon (Cascade Lake) Platinum 8269CY, 16 vCPU, 64GiB memory), and NVIDIA GeForce RTX 2080 Ti, covering the backends of AVX256 and AVX512 with 4 threads, and CUDA, respectively. We take ResNet 18 [22], ResNet 50 [22], MobileNet V2 [38], SqueezeNet V1.1 [27], ShuffleNet V2 [33], BERT-SQuAD 10 [10], and DIN [46], as the testing models, which are commonly used in CV, NLP, and recommendation applications. The input size of CV models is set to $1 \times 3 \times 224 \times 224$, the input size of BERT-SQuAD 10 is set to $(1 \times 256, 1 \times 256, 1 \times 256, 1)$, while the input size of DIN is set to $1 \times 100 \times 32$. We show the inference time of the CV and NLP models in the left part of Figure 10, and omit the results of DIN, which are quite low (e.g., less than 0.2ms on iPhone 11 using MNN). We can observe that MNN significantly outperforms TensorFlow (Lite) and PyTorch (Mobile) in almost all the test cases. Besides higher performance, MNN is also more full-featured on the side of mobile devices, given that MNN can support all the models on each device-side backend, whereas TensorFlow Lite and PyTorch Mobile fail to support some backends and/or models.

We continue to compare MNN with TVM. We take MacBook Pro 2019 and NVIDIA GeForce RTX 2080 Ti as the host machines of TVM to do auto-tuning and compiling for the mobile devices and the GPU server, respectively. The number of trials in TVM auto-tuning is set to 30. Since TVM auto-tuning for BERT-SQuAD 10 on two mobile devices incurs timeout crash, we take the default parameter settings for model inference. From the evaluation results depicted in the right part of Figure 10, one key observation is that the auto-tuning and the compiling of TVM roughly cost thousands of seconds. In contrast, the semi-auto search of MNN for runtime optimization costs roughly hundreds of milliseconds. Further incorporating the comparative analysis in Section 4.1, we can draw that MNN can support the industrial scenarios that involve numerous heterogeneous devices and require frequent and quick task iteration, whereas TVM cannot. The second key observation is that the inference time of MNN is lower than TVM for each model on each backend, especially on the GPU server. Such superiority is mainly due to the manual operator-level, backend-level optimization in MNN.

We next compare Walle's Python thread-level VM with the original Python VM (i.e., CPython with GIL) using roughly 30 million online ML task executions. We define performance as the reciprocal of task execution time and show the average performance improvement in Figure 11. For the light-weight, middle-weight, and heavy-weight tasks, Python thread-level VM gains 52.11%, 144.36%, and 25.70% of performance improvement, respectively. We can draw that task-level multi-threading without GIL is the key of performance boosting.

We finally evaluate the latency of the real-time tunnel over roughly 364 million uploads. Figure 12 shows the latency and the number of uploads for varying data sizes. We can observe that more than 90% uploads are under 3KB with less than 250ms on average. Even when the sizes of 0.1% uploads grow to 30KB, the average delay increases only to around 450ms.

7.3 Deployment Platform Statistics

The deployment platform in Walle has supported 30+ mobile APPs (e.g., Mobile Taobao, AliExpress, Xianyu, Youku, Cainiao Guoguo) in Alibaba since the end of 2017, running for roughly 1,500 days. It has deployed 1,000+ kinds of ML tasks in total, each with 7.2 versions on average. Currently, the deployment platform is maintaining and monitoring 348 kinds of active tasks on more than 0.3 billion mobile devices. To demonstrate the timeliness of task deployment, we randomly select an ML task, monitor its release process, and depict in Figure 13 how the number of covered devices changes as the elapsed time grows. The first segment of the curve shows the gray release stage, which takes 7 minutes to cover all the 6 million online devices. In particular, roughly 4 million devices are incrementally covered within the last minute. Then, the number of covered devices increases as more mobile devices become online. Until 19 minutes later, almost 22 million devices have been covered. The statistics show the scalability and timeliness of the deployment platform in Walle.

8 Related Work

In this section, we briefly review some related work in both academia and industry.

Cloud-Based ML System. Many companies have built their ML systems on the cloud, which are backed by their

cloud computing platforms, such as Amazon Web Services, Microsoft Azure, Alibaba Cloud, and Google Cloud. The architecture is clear and comprises the standard modules of data storage (e.g., HBase [14] and HDFS [13]), batch and stream processing (e.g., Storm [42], Spark [45], and Flink [7]), ML engines (e.g., TensorFlow [1], PyTorch [36], and MXNet [8]), virtualization and containerization (e.g., KVM [37] and docker [26]), and elastic orchestration (e.g., Kubernetes [15]).

On-Device ML System. Some modules are open source with rapid development in terms of well balancing light weight, necessary functionality, and high performance, including on-device inference engines (e.g., TensorFlow Lite [16], PyTorch Mobile [12], Core ML [3], and NCNN [39]); and SQLite [24], which is a small and self-contained SQL database engine. However, the whole architecture is still in the dark, and several core capabilities are absent, such as an on-device execution environment that supports quick development and concurrent execution of multiple ML tasks, and light-weight data processing and model training libraries for diverse CV, NLP, and recommendation tasks.

Device-Cloud Collaborative ML. The concept can stretch back to edge/mobile computing, but focuses on the collaboration of the cloud and mobile devices in executing complex ML tasks, rather than offloading simple data analysis tasks from the cloud to edge servers or mobile devices. Previous work focused on the algorithmic framework or solution and was normally specific to a certain kind of application. In contrast and in parallel, Walle targets at the general-purpose and large-scale production system support. We review some representative work as follows.

An initial paradigm is to keep model training on the cloud but offload model inference (e.g., facial recognition, photo beautification, and question answering) to mobile devices, validating the on-device advantages in reducing latency and protecting privacy. The key of this paradigm's proliferation is the advances of model compression algorithms to reduce model size and optimize model structure, such as quantization [19], pruning and sparification [20], knowledge distillation [23], and neural architecture search [47]. Later, Mistify [18] automated the cloud-to-device model porting process given the customized requirements of heterogeneous mobile devices, while some work designed more reasonable task splitting strategies rather than offloading the full inference task. For example, Neurosurgeon [29] was proposed to automatically partition DNN computation between a mobile device and the cloud at the granularity of DNN layers.

Besides inference, the popular cross-device federated learning (FL) framework [34] elegantly generalizes the conventional parameter server framework [30] and enables multiple mobile devices to collaboratively train a global model under the coordination of a cloud server. The tenet of FL is to keep user data on local devices, thereby protecting data security and privacy. The device-cloud collaboration in FL is purely through exchanging model and its update periodically. The task splitting strategy is that mobile devices conduct model training, and the cloud aggregates model updates. Google has experimentally deployed FL on its Android keyboard, called Gboard, to polish language models [5].

Finally, many application-specific solutions were proposed under the principle of device-cloud collaboration. FilterForward [6] and Reducto [31] considered how to effectively and efficiently do camera-side frame filtering with ML techniques to facilitate cloud-side video analytics. DDS [11] adopted an interactive workflow, where a camera first uploads a low-quality video stream and re-sends a few key regions with higher quality according to the cloud's feedback to improve inference accuracy. COLLA [32] studied the user behavior prediction task with RNN and leveraged knowledge distillation to mutually and continuously transfer the knowledge between the device-side small models and the cloud-side large model, thereby mitigating data heterogeneity and data drift over time. DDCL [44] and CoDA [17] focused on recommendation. DDCL relied on patch learning for on-device model personalization and adopted model distillation to integrate the patches from mobile devices into the cloud-side global model. CoDA, instead, was proposed to retrieve similar samples from the cloud's global pool to augment each mobile device's local dataset for training personalized recommendation models. Backed by Walle, CoDA was deployed in Mobile Taobao.

9 Conclusion

In this work, we have built the first end-to-end, general-purpose, and large-scale production system, called Walle, for device-cloud collaborative ML. Walle is oriented by the lifecycle of ML tasks and consists of a cross-platform, high-performance, and quickly iterative compute container; a more reasonable and efficient data pipeline; and a scalable, timely, and robust deployment platform. Evaluation of Walle in practical e-commerce scenarios and extensive micro-benchmarks have demonstrated the necessity of device-cloud collaboration and the superiority of each ingredient. Walle has been deployed in Alibaba for wide scale production use, serving billion-scale users with mobile devices every day.

Acknowledgments

We sincerely thank our shepherd, Wenjun Hu, for her insightful and thorough guidance. We thank the anonymous OSDI reviewers for their constructive feedback. We thank Kai Liu, Hansong Liu, Hao Jiang, Zhijie Cao, and Yan Chen from Alibaba for their great support. This work was supported in part by National Key R&D Program of China No. 2019YFB2102200, China NSF grant No. 62025204, 62072303, 61972252, 61972254, 61832005, and 62141220, and Alibaba Innovation Research (AIR) Program. The opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies or the government.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[2] Alibaba. MNN, 2019. https://github.com/alibaba/MNN.

[3] Apple. Core ML, 2017. https://developer.apple.com/machine-learning/core-ml/.

[4] Apple. App store review guidelines: Section 2.5.2, 2021. https://developer.apple.com/app-store/review/guidelines/#software-requirements.

[5] Kallista A. Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In *Machine Learning and Systems (MLSys)*, 2019. https://proceedings.mlsys.org/book/271.pdf.

[6] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G. Andersen, Michael Kaminsky, and Subramanya Dulloor. Scaling video analytics on constrained edge nodes. In *Proceedings of Machine Learning and Systems (MLSys)*, 2019. https://proceedings.mlsys.org/book/273.pdf.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015. http://sites.computer.org/debull/A15dec/p28.pdf.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv: 1512.01274*, 2015. https://arxiv.org/pdf/1512.01274.pdf.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018. https://www.usenix.org/system/files/osdi18-chen.pdf.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018. https://arxiv.org/pdf/1810.04805.pdf.

[11] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, pages 557–570, 2020. https://dl.acm.org/doi/pdf/10.1145/3387514.3405887.

[12] Facebook. PyTorch Mobile, 2019. https://pytorch.org/mobile/home/.

[13] Apache Software Foundation. Apache Hadoop, 2006. https://hadoop.apache.org/.

[14] Apache Software Foundation. HBase, 2008. https://hbase.apache.org/.

[15] Google. Kubernetes, 2014. https://kubernetes.io/.

[16] Google. TensorFlow Lite, 2017. https://www.tensorflow.org/lite.

[17] Renjie Gu, Chaoyue Niu, Yikai Yan, Fan Wu, Shaojie Tang, Rongfeng Jia, Chengfei Lv, and Guihai Chen. On-device learning with cloud-coordinated data augmentation for extreme model personalization in recommender systems. *arXiv: 2201.10382*, 2022. https://arxiv.org/pdf/2201.10382.pdf.

[18] Peizhen Guo, Bo Hu, and Wenjun Hu. Mistify: Automating DNN model porting for on-device inference at the edge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 705–719, 2021. https://www.usenix.org/system/files/nsdi21-guo.pdf.

[19] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning (ICML)*, pages 1737–1746, 2015. http://proceedings.mlr.press/v37/gupta15.html.

[20] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 1135–1143, 2015. https://proceedings.neurips.cc/paper/2015/hash/ae0eb3eed39d2bcef4622b2499a05fe6-Abstract.html.

[21] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. https://www.nature.com/articles/s41586-020-2649-2.pdf.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. https://doi.org/10.1109/CVPR.2016.90.

[23] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *arXiv: 1503.02531*, 2015. https://arxiv.org/pdf/1503.02531.pdf.

[24] Dwayne Richard Hipp. SQLite, 2000. https://www.sqlite.org/.

[25] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv: 1704.04861*, 2017. https://arxiv.org/pdf/1704.04861.pdf.

[26] Solomon Hykes. Docker, 2013. https://www.docker.com/.

[27] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360*, 2016. https://arxiv.org/pdf/1602.07360.pdf.

[28] Intel. OpenCV, 2000. https://opencv.org/.

[29] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor N. Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 615–629, 2017. https://dl.acm.org/doi/pdf/10.1145/3037697.3037698.

[30] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 583–598, 2014. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf.

[31] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, pages 359–376, 2020. https://dl.acm.org/doi/pdf/10.1145/3387514.3405874.

[32] Yan Lu, Yuanchao Shu, Xu Tan, Yunxin Liu, Mengyu Zhou, Qi Chen, and Dan Pei. Collaborative learning between cloud and end devices: An empirical study on location prediction. In *ACM/IEEE Symposium on Edge Computing (SEC)*, pages 139–151, 2019. https://dl.acm.org/doi/pdf/10.1145/3318216.3363304.

[33] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In *European Conference on Computer Vision (ECCV)*, pages 122–138, 2018. https://doi.org/10.1007/978-3-030-01264-9_8.

[34] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1273–1282, 2017. http://proceedings.mlr.press/v54/mcmahan17a.html.

[35] NVIDIA. Data layout formats, 2022. https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#data-layout-formats.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie

Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 8024–8035, 2019. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[37] Qumranet, Inc. Kernel-based virtual machine, 2007. http://www.linux-kvm.org/.

[38] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018. https://doi.org/10.1109/CVPR.2018.00474.

[39] Tencent. NCNN, 2017. https://github.com/Tencent/ncnn/.

[40] Zhi Tian, Chunhua Shen, Hao Chen, and Tong He. FCOS: Fully convolutional one-stage object detection. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9626–9635, 2019. https://ieeexplore.ieee.org/document/9010746.

[41] Linus Torvalds. Git, 2005. https://git-scm.com/.

[42] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. Storm@Twitter. In *ACM International Conference on Management of Data (SIGMOD)*, pages 147–156, 2014. https://dl.acm.org/doi/pdf/10.1145/2588555.2595641.

[43] Guido van Rossum. CPython, 1994. https://github.com/python/cpython.

[44] Jiangchao Yao, Feng Wang, Kunyang Jia, Bo Han, Jingren Zhou, and Hongxia Yang. Device-cloud collaborative learning for recommendation. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 3865–3874, 2021. https://dl.acm.org/doi/pdf/10.1145/3447548.3467097.

[45] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010. https://www.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf.

[46] Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1059–1068, 2018. https://dl.acm.org/doi/pdf/10.1145/3219819.3219823.

[47] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017. https://openreview.net/pdf?id=r1Ue8Hcxg.

Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization

Colin Unger[†♠] Zhihao Jia[‡♭♠] Wei Wu[*◇] Sina Lin[§] Mandeep Baines[♭]
Carlos Efrain Quintero Narvaez[♭] Vinay Ramakrishnaiah[*] Nirmal Prajapati[*]
Pat McCormick[*] Jamaludin Mohd-Yusof[*] Xi Luo[♯] Dheevatsa Mudigere[♭]
Jongsoo Park[♭] Misha Smelyanskiy[♭] Alex Aiken[†]

Stanford University[†] *Carnegie Mellon University*[‡] *Los Alamos National Lab*[*]
NVIDIA[◇] *Microsoft*[§] *Meta*[♭] *SLAC National Accelerator Laboratory*[♯]

Abstract

This paper presents Unity, the first system that jointly optimizes algebraic transformations and parallelization in distributed DNN training. Unity represents both parallelization and algebraic transformations as substitutions on a unified *parallel computation graph* (PCG), which simultaneously expresses the computation, parallelization, and communication of a distributed DNN training procedure.

Optimizations, in the form of graph substitutions, are automatically generated given a list of operator specifications, and are formally verified correct using an automated theorem prover. Unity then uses a novel hierarchical search algorithm to jointly optimize algebraic transformations and parallelization while maintaining scalability. The combination of these techniques provides a generic and extensible approach to optimizing distributed DNN training, capable of integrating new DNN operators, parallelization strategies, and model architectures with minimal manual effort.

We evaluate Unity on seven real-world DNNs running on up to 192 GPUs on 32 nodes and show that Unity outperforms existing DNN training frameworks by up to 3.6× while keeping optimization times under 20 minutes. Unity is available to use as part of the open-source DNN training framework FlexFlow at https://github.com/flexflow/flexflow.

1 Introduction

Deep neural networks (DNNs) are becoming progressively larger and computationally more expensive to train, and as they have grown, so has interest in optimizing their execution to reduce training times and improve scalability. Two key classes of optimizations shown to yield significant performance improvements across diverse model architectures are algebraic transformations and parallelization.

Algebraic transformations exploit operator identities to perform the underlying computation in a more efficient way, but ignore parallelization and distribution of training. Common examples of algebraic transformations include operator fusion, which merges two operators into a single semantically-equivalent operator whose computation is more efficient, and operator reordering, where the associativity or commutativity of sets of operators allows them to be reordered into more efficient configurations or to expose further optimization opportunities. More explanation of algebraic transformations, along with examples, is provided in Section 2.2.

Parallelization, in contrast, distributes operators over multiple devices, but does not change the way in which the underlying computation is performed. DNN training exploits a class of parallelism named *partition-n-reduce* [59], in which every distributed subcomputation of an operator must perform the same computation, and may only differ in the input data it consumes. The tensor computations in DNN training are particularly well-suited to this form of parallelism, and many *parallelism dimensions* along which to divide distributed operators have been identified, such as data [6], model [13], spatial [27], reduction [50], and pipeline [39]. For a detailed overview of these various approaches, see Section 2.1.

When applied effectively, these two techniques can improve training times by more than an order of magnitude. However, effective application is nontrivial. Rewriting the computation graph for maximum speedup can require many transformations, some of which may harm performance except in the context of a longer sequence of transformations [26]. The optimal parallel execution strategy for a model often requires simultaneously exploiting multiple parallelization dimensions and using different parallelization schemes for each operator [24]. Early work relied on the programmer to manually determine the correct optimizations to apply [6]. While manual optimization allows fine-grained control over the model's performance, it requires many hours of tuning by experts to achieve good performance. As the pace of new developments in model design has increased, manual optimization has struggled to scale beyond the most commonly used models.

Recent work has focused on automating optimizations. MetaFlow [26], TASO [25], and PET [58] propose algorithms for automatically generating and applying algebraic transformations by posing optimization as a search problem.

♠ Contributed equally.

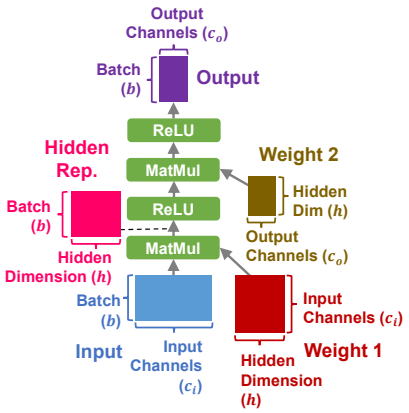
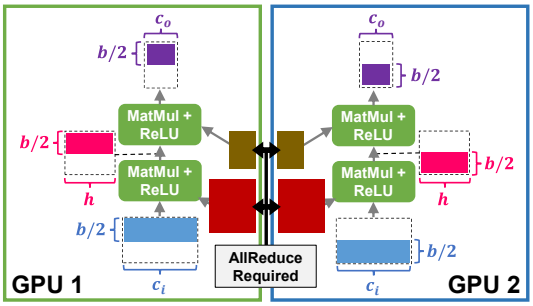


Figure 1: Computation graph for a 2-layer MLP.

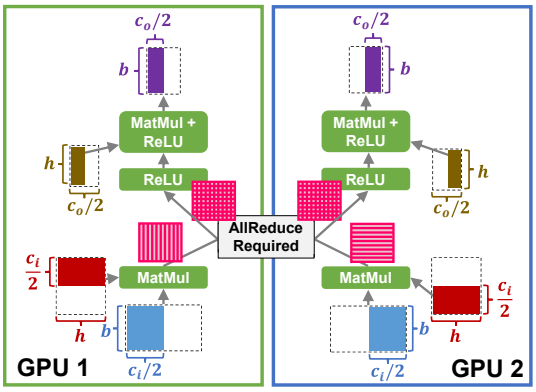
FlexFlow [27], automap [48], Tofu [59], and Whale [23] bring a similar approach to parallelism. These works present impressive benchmarks, yielding the impression that automating algebraic and parallelization optimization is a solved problem.

However, to reduce training time as much as possible, we want to apply both of these optimizations, but the most effective way to combine algebraic and parallelization optimizations is not obvious. The simplest solution is to apply them independently, in one of two orders: algebraic optimization followed by parallelization, or the reverse. The reverse order turns out to be problematic: since algebraic transformations can introduce new operations or replace existing ones, running algebraic optimization after parallelizations have been assigned can lead to the final solution having operations without assigned parallelizations (if the operation was created) or invalid parallelizations (if the operation was replaced). Workarounds can be used to fix invalid solutions by using default parallelization strategies or copying the strategies of nearby operators, but it is easy to find cases in which these workarounds lead to suboptimal solutions. As such, applying algebraic optimization before parallelization is the only option, but as we see in the next example, it can miss significant optimization opportunities.

Consider the computation graph shown in Figure 1, which represents a 2-layer multilayer perceptron (MLP). If we are optimizing independently (also referred to as "sequentially"), we start by applying algebraic transformations without considering parallelism. A typical algebraic optimizer will fuse the `MatMul` and `ReLU` operators to remove redundant memory loads and stores. The model is then parallelized (we consider only 2 GPUs for simplicity) resulting in Figure 2a: data parallelism is used for both operators and thus the weight gradients must be synchronized with an AllReduce. Since weight 1 has size $c_i h$ and weight 2 has size hc_o, the total communication is $2(c_i h + hc_o)$. Using a set of parameters for a basic image classification model for MNIST ($b = 64$, $h = 512$, $c_o = 10$, $c_i = 28 \times 28 = 784$) yields a total communication of $813,056d$ bytes, where d is the element size.



(a) Sequential optimization.



(b) Joint optimization.

Figure 2: Comparing joint and sequential optimizations.

Instead of independently applying algebraic transformations and parallelization, we can combine them and solve a single joint optimization problem that discovers the solution in Figure 2b. By not fusing the first `MatMul` and `ReLU`, more efficient reduction parallelism can be used. This requires synchronizing the activation and gradient of the first `MatMul`'s output, but not the weights: a total inter-GPU communication of $4bh$, or $131,072d$ bytes for our MNIST example. Joint optimization reduces communication by $6\times$, which far exceeds the cost of not fusing the first `ReLU`.

As this example shows, joint optimization is necessary to maximize performance. However, it also poses significant challenges. The first is representation: existing frameworks perform optimizations on a model's computation graph. As discussed above, algebraic transformations can leave operators in the computation graph with unassigned or invalid parallelizations. To prevent such invalid solutions from arising during search, we need a representation that allows algebraic transformations to consider the current parallelization before being applied. Further discussion of the representation challenges is in Section 3.4.

The second challenge is scalability: existing search-based approaches already struggle to scale up to large models and GPU counts. Improvements have been made for algebraic transformations alone [62], but the complexity of these solutions makes adding parallelization a daunting task. Simultane-

ously considering both optimization classes only exacerbates this problem by exponentially increasing the search space size. For joint optimization to be practical, search algorithms must improve on the scalability of past techniques.

1.1 Unity's Approach

The key idea behind Unity is to represent both algebraic transformations and parallelization as graph substitutions on a unified *parallel computation graph*, and then to use a hierarchical search algorithm to efficiently identify which combination of substitutions yields the best performance. Figure 3b shows an overview of Unity, which differs from existing frameworks in the following ways:

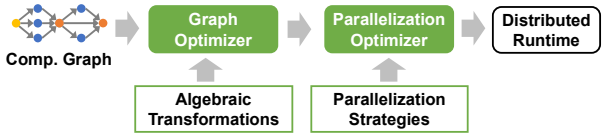
Unified graph representation. We introduce the *parallel computation graph* (PCG)[1] as a unified representation of distributed DNN training that simultaneously expresses computation, parallelism, and data movement. All parallelization strategies used in existing frameworks can be represented as specific PCGs, and parallelization and algebraic transformations as sequences of graph substitutions. pONNX [57] previously proposed merging computation and parallelism into a single graph, but certain design decisions prevent Unity-style joint optimization. For a detailed comparison, see Section 3.4.

Transformation generation and verification. Unity does not require users to explicitly define possible parallelization strategies for DNN training. Unlike prior work that automatically generates parallelization strategies [59] or algebraic transformations [25], by using the PCG Unity is able to generate both kinds of transformations with a single approach, as well as hybrid algebraic-parallelization optimizations absent in prior automated approaches. Automatically generating and verifying transformations greatly reduces the engineering effort required to support different parallelism dimensions and enables extensibility to new operators.

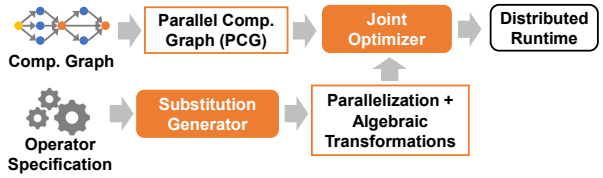
Joint optimization. Unity uses a hierarchical search algorithm to discover highly optimized PCG substitutions and device placements while maintaining scalability to models with hundreds of operators distributed over hundreds of GPUs. Unity's cost model includes both computation and communication time, and the search algorithm handles custom network topologies and heterogeneous compute devices. Despite the exponentially larger search space being considered, Unity outperforms existing search-based approaches (see Section 6).

The rest of this paper provides additional background (Section 2), discusses Unity's design and implementation (Sections 3, 4, and 5), and evaluates its performance on seven real-world DNNs (Section 6). For widely-used DNNs highly optimized by existing frameworks, such as BERT [14], Unity matches the performance of existing expert-designed strategies while being completely automated. For complex DNN architectures with a mixture of compute- and communication-intensive operators, such as DLRM [41] and CANDLE-Uno [1], Unity is up to 3.6× faster than existing frameworks.

[1]To prevent ambiguity, we use the term *computation graph* strictly to refer to the conventional computation graph used in prior work, and *parallel computation graph* or PCG to refer to Unity's new unified representation.



(a) Existing approach.



(b) Unity's approach.

Figure 3: Comparing existing DNN frameworks and Unity.

2 Background

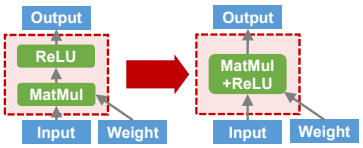
We first provide a brief overview of the two classes of optimizations that Unity exploits, parallelization (Section 2.1) and algebraic transformations (Section 2.2), as well as a discussion of how they are represented in existing systems (Section 2.3). For a discussion of how Unity interacts with other classes of optimizations, see Section 8.

2.1 Parallelization

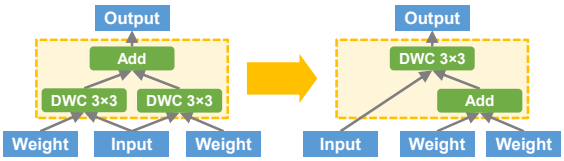
The massively parallel nature of tensor algebra creates many opportunities for parallelizing DNN training. We identify six primary forms of parallelism leveraged in DNN systems:

1. *Data parallelism* is the most common approach used in existing frameworks [6, 9, 42]. Data parallelism keeps a replica of the entire DNN model on every device and assigns each a subset of the training data.
2. *Model parallelism* divides a DNN model into disjoint sub-models and trains each sub-model on a dedicated device.
3. *Spatial parallelism*[2] divides the spatial dimensions of a tensor (e.g., the height and width of images) into multiple partitions, each of which is assigned to a specific device [27]. Spatial parallelism often requires synchronizing the shared elements (e.g., the shared pixels along the boundary of different sub-images) between devices.
4. *Reduction parallelism* exploits the linearity of tensor algebra operators. For a matrix multiplication $C = A \times B$, reduction parallelism splits A along its columns and B along its rows as follows: $A = [A_1, \ldots, A_n]$, $B = [B_1^T, \ldots, B_n^T]^T$ The matrix multiplication is distributed across n devices, with the i-th device computing $C_i = A_i \times B_i$. An extra reduction afterward recovers the original result: $C = \sum_i C_i$.

[2]Spatial parallelism was called *attribute parallelism* in [27].



(a) Basic operator fusion.



(b) A more complex algebraic transformation.

Figure 4: Example algebraic transformations. `DWC` stands for `DepthwiseConv` (i.e., depth-wise separable convolution).

5. *Pipeline parallelism* exploits the opportunity to parallelize across different training iterations [39].
6. *Operator-specific parallelism.* The introduction of new DNN operators provides operator-specific parallelization opportunities. For example, the following equation shows the batched matrix multiplication used in Transformer [54]: $output(s,h,o) = \sum_i input(s,h,i) \times weight(h,o,i)$. This differs from typical matrix multiplication in that it applies a different weight for each input sample. As a result, these batched matrix multiplications across attention heads can be run in parallel (i.e., the h dimension) without any tensor replication or synchronization.

Most parallelizations are not pure performance optimizations, but are instead trade-offs among different cost metrics. For example, applying data parallelism reduces per-device computation time at the cost of increased memory usage and data movement for storing and synchronizing model parameters. Thus, DNN operators typically require a combination of these forms of parallelism to achieve optimal performance.

2.2 Algebraic Transformations

Algebraic transformations are very diverse and are not as easily categorized as the forms of parallelism, so we instead provide examples. For a more comprehensive exploration of algebraic transformations, see [25].

The most basic algebraic transformation is operator fusion, shown in Figure 4a. Unfused, the device needs to load and store activations to and from memory twice, once before and after each operator. If the two operators are fused, however, the combined kernel can compute the `ReLU` operation as it stores the outputs of the `MatMul` back to memory.

For a more complex example, see Figure 4b. By exploiting `DepthwiseConv`'s linearity, a computation that previously required two `DepthwiseConv` operations now only requires one plus an additional `Add`, effectively halving the amount of computation needed.

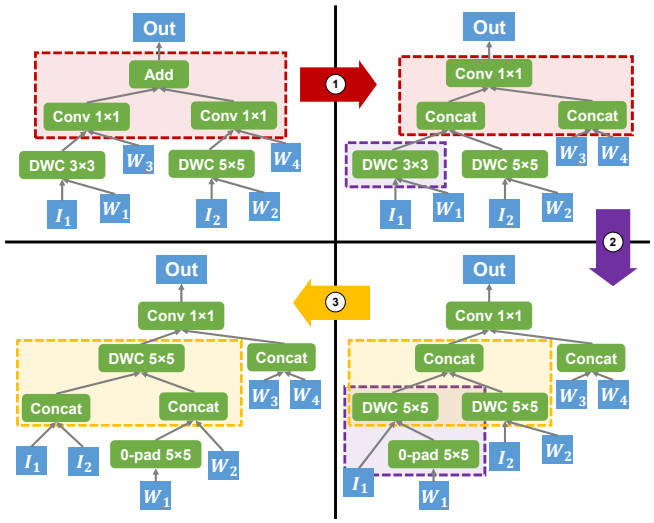


Figure 5: Compositions of small algebraic transformations can lead to significant changes.

Small algebraic transformations can be composed to create large changes. Consider the sequence of transformations shown in Figure 5: while each individual transformation is relatively small, the final output is radically different from the original computation graph. Also, notice that not all performance gains are realizable in a single transformation: for example, moving from graph 2 to graph 4 reduces the amount of computation by reducing the number of `DepthwiseConv` operations performed, but it is first necessary to pass through graph 3 which performs worse than either graph 2 or 4.

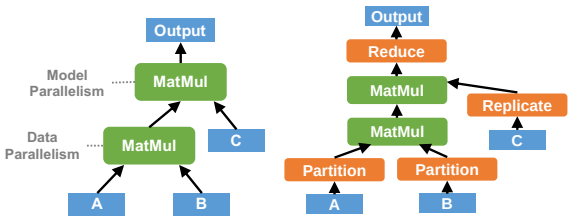
2.3 Intermediate Representations

Most existing optimizing frameworks represent a DNN architecture as a *computation graph*[3]: a node is a mathematical tensor operator (e.g., matrix multiplication, etc.), and an edge is a tensor (i.e., n-dimensional array) passed between operators. An example computation graph is shown in Figure 6a. Algebraic transformations are performed by iteratively applying graph substitutions, and the model is parallelized by assigning each node a set of parallelism annotations.

This representation has two limitations. First, while using distinct representations for algebraic transformations (i.e., graph substitutions) and parallelization (i.e., node annotations) is convenient, it hinders joint optimization. The key issue is that algebraic transformations can add or replace nodes in the graph, while parallelization views the computation graph as static and thus cannot handle these newly-created, unannotated nodes. This prevents interleaving the two search algorithms, since at any time a substitution can transform a valid parallelization into an invalid one.

Second, a computation graph does not explicitly capture

[3] Alternative representations are discussed in Section 3.4 and Section 7.



(a) Computation graph. (b) Parallel computation graph.

Figure 6: Comparing computation graph and PCG. Both graphs describe the same parallelization of two consecutive matrix multiplications $(A \times B) \times C$ (a simplified form of attention). The green and orange boxes denote regular DNN operators and Unity's new parallelization operators (see Section 3.3) respectively.

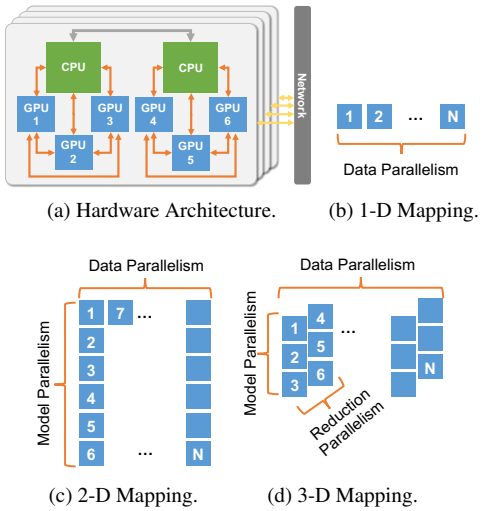
the communication costs associated with parallelism. This absence makes it difficult for algebraic transformations to reason about the impact on the performance of the final model.

3 Parallel Computation Graph

To solve the shortcomings of the existing model representations described in Section 2.3, we introduce the *parallel computation graph* (PCG) as a unified representation of distributed DNN training that is capable of simultaneously expressing computation, parallelism, and communication. The PCG allows Unity to consider both algebraic transformations and parallelization as graph substitutions on a common graph. While the PCG is not the first to merge computation and parallelization into a single graph, the PCG is tailored for optimization and as such differs from prior unified graph representations in key aspects, which we discuss in Section 3.4.

PCGs extend the existing computation graph representation by allowing nodes to represent changes in parallelization in addition to mathematical tensor operations, and edges to represent distributed movement of tensor data in addition to data dependence. A set of *parallelization operators* are added that allow PCGs to express all existing parallelization strategies and provide an explicit representation of data movement and its associated costs during training. Additionally, each operator in a PCG is associated with a *machine mapping*, denoting how the execution of the operator is mapped to individual processors in a parallel machine. Figure 6b shows an example of a PCG.

Sections 3.1, 3.2, and 3.3 provide a brief description of the tensor representation, machine mappings, and parallelization operators, respectively. Finally, Section 3.4 discusses the design decisions that make the PCG uniquely suited for joint optimization, and how it differs from alternative unified graph representations.



(a) Hardware Architecture. (b) 1-D Mapping.

(c) 2-D Mapping. (d) 3-D Mapping.

Figure 7: Example machine mapping for a compute node in our evaluation. (a) shows the node's hardware architecture, where and orange and grey arrows denote NVLink and X-Bus. Numbers in mapping examples denote GPU ids.

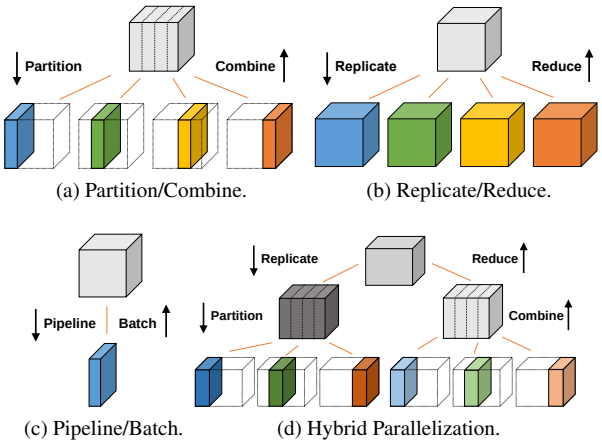
3.1 Tensor Representation

Unity models tensors as a set of data dimensions, each of which has two fields: a size and a degree. The degree field specifies the number of partitions the tensor has been divided into along that dimension. Every tensor also includes a special *replica dimension*, which represents the number of replicas of that tensor's data.

3.2 Machine Mappings

Each operator in a PCG is associated with a *machine mapping*, an n-dimensional array of devices/processors that specifies on which device to run each piece of the operator's computation. More formally, given an operator and a set of n applicable parallel dimensions with degrees $d_1, \ldots, d_n$, Unity divides the operator into $d_1 \times d_2 \times \ldots \times d_n$ parallel tasks, which we reference with tuple indices of the form $(i_1, \ldots, i_n)$ where $0 \leq i_k < d_k$. A machine mapping is a map from task indices $(i_1, \ldots, i_n)$ to individual GPUs that will be used to run that parallel task. For convenience, we also define the machine mapping of an entire PCG to be the set of machine mappings of each of its constituent operators.

Figure 7 shows some example machine mappings for the Summit compute nodes [55] used in our evaluation. The hardware architecture is depicted in Figure 7a. Figure 7b shows a basic 1-D machine mapping for data parallelism, while Figure 7c shows a 2-D machine mapping of a hybrid parallelization strategy combining data and model parallelism, where model parallelism is applied across GPUs within the same compute node and data parallelism across distinct compute nodes. Figure 7d shows a 3-D machine mapping where we apply model parallelism across GPUs attached to the same



(a) Partition/Combine. (b) Replicate/Reduce.

(c) Pipeline/Batch. (d) Hybrid Parallelization.

Figure 8: Parallelization operators in Unity.

CPU, reduction parallelism across GPUs attached to different CPUs but on the same compute node, and data parallelism across different compute nodes.

Unity includes a comprehensive set of machine mappings that capture effective usages of a parallel machine. In addition, developers can register custom machine mappings tailored to specific hardware architectures. For example, when node pairs in a cluster have different network bandwidths and latencies, an extra dimension can be added to the existing machine mappings to represent node-level locality.

Machine mappings provide two key desirable properties: *expressiveness* and *scalability*. All effective distributions of parallel tasks in a PCG can be captured in just a few machine mappings, and complex features of a machine's hardware architecture can be easily leveraged through adding additional machine mappings. Machine mappings also allow Unity to capture all effective device assignments while remaining linear in the number of devices and aid Unity's search algorithm by removing inefficient assignments from consideration.

3.3 Parallelization Operators

Unity uses six parallelization operators to capture the computation and communication costs associated with different parallelization strategies. These six are further divided into three pairs, where one operator is the "back propagation" of the other (e.g., when back propagation is done on `Partition` it becomes semantically equivalent to `Combine`, and the same in reverse). The three pairs are:

1. *Partition and Combine*: `Partition` and `Combine` change a tensor's degree of parallelism. More specifically, `Partition` increases the parallelism degree of a tensor dimension by splitting the dimension into multiple equal-sized partitions, as shown in Figure 8a. `Combine` performs the reverse: reducing a tensor's degree of parallelism by concatenating multiple partitions into one.
2. *Replicate and Reduce*: `Replicate` and `Reduce` control the parallelism degree of the replica dimension by copying and summing tensors, as shown in Figure 8b. Parameter synchronization is naturally captured as the back propagation of `Replicate` operations applied to weight tensors.
3. *Pipeline and Batch*: `Pipeline` splits a tensor dimension into equal size partitions and processes one partition at a time, while `Batch` aggregates tensors across iterations (see Figure 8c). Note that `Pipeline` does not modify the parallelism degree of a tensor dimension, but instead reduce its size.

As a basic demonstration of the PCG's expressiveness, Figure 9 illustrates how Unity's six parallelization operators can represent some example parallelization strategies from Section 2.1. These parallelization operators can also be composed to create hybrid parallelism. Figure 8d shows an example that applies `Replicate` and `Partition` on the same tensor dimension, replicating the tensor and partitioning each replica. To improve efficiency, Unity replaces particular sequences of parallelization operators with fused versions at run time (e.g., a `Reduce` followed by a `Replicate` can be implemented as an AllReduce).

3.4 Discussion and Comparison

Unity's decision to use the PCG instead of an annotated computation graph is driven by how easily the representations lend themselves to joint search and not a fundamental limitation of annotated computation graphs. Theoretically, there exist annotation languages isomorphic to the PCG, but attempts to design such a language quickly lead to a number of difficulties.

First, because each operator can use different forms of parallelism, including operator-specific forms of parallelism, the number of annotations quickly grows prohibitively large. Which annotations are supported by which operators, along with their semantics and composition, must then be baked into the representation itself. By comparison, Unity's PCG moves this knowledge into the PCG substitutions, which are generated automatically. This separation of concerns makes the core of Unity simpler and easier to maintain.

Second, not explicitly representing communication forces communication patterns along dataflow edges to be reconstructed from their source and destination node annotations, which is difficult due to the expressive forms of parallelism Unity considers. Specifically, supporting n parallelism dimensions requires considering up to 2^n different subsets of these dimensions and thus $2^n \times 2^n = 4^n$ potential communication patterns between operators. Unity explicitly represents communication patterns throughout search, obviating the need for a complex analysis to reconstruct them. Representing these patterns via a small set of parallelization operators also allows Unity to easily recognize and optimize common communication patterns, such as executing a pair of `Reduce-Replicate`

operators as an AllReduce. In an annotated computation graph, these optimizations become entangled with the code for reconstructing the communication patterns themselves, adding significant complexity and implementation effort.

Finally, jointly optimizing an annotated computation graph is challenging, as algebraic transformations can introduce new operators which, since they have not yet been parallelized, lack annotations. As such, the internal representation becomes underspecified and the cost becomes undefined. It is possible to add an additional mechanism to "fill in" these missing annotations such as inserting a random annotation, a fixed value, a value from a neighboring node (though this becomes challenging when neighboring nodes have differing parallelizations), or evaluating the valid parallelizations and choosing the best one. However, Unity's PCG avoids this additional complexity by representing each parallelization strategy for the new operator as one or multiple PCG substitutions, offering an efficient and uniform approach to joint optimization.

pONNX. Unity is not the first to integrate computation and parallelism into a single graph: pONNX [57] proposed doing so using `Split`, `Concat`, and custom operators `Send` and `Recv`. However, Unity focuses on optimization while pONNX is designed as a serialization format, leading to critical differences.

First, an operator in pONNX with a parallelism degree of n is duplicated n times, requiring an optimizer to reconstruct the operator from multiple nodes. Unity simply adds a parallelism operator so the operator remains a single node in a PCG.

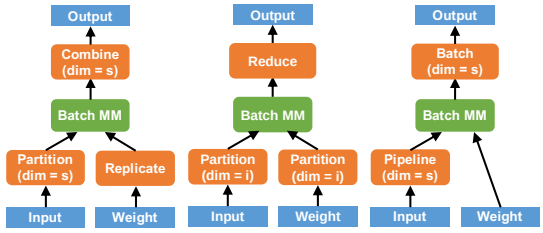
Second, pONNX assigns every communication its own `Send`/`Recv` node, which dramatically increases the size of the graph. Since communication patterns in DNN training are highly regular, Unity eschews materializing every communication in favor of optimizing communication patterns (e.g., `Reduce`, `Replicate`, etc.), which allows Unity to represent communication costs without reasoning about individual communications.

Finally, pONNX makes device placement part of the operator, while Unity represents it separately as a machine mapping. This allows Unity's search to optimize device assignments separately and to ignore the symmetries created by a large number of compute devices with identical capabilities.

Additional unified representations have been proposed [47, 48], which are discussed in Section 7.

4 Graph Substitutions

Since Unity represents both algebraic transformations and parallelization as graph substitutions, the effectiveness of joint optimization relies on having an appropriate set of graph substitutions. The number of potential substitutions increases exponentially with size, so Unity represents large and complex algebraic transformations and parallelization strategies as compositions of small PCG substitutions. For example, Figure 10 shows the sequence of substitutions for the hand-tuned parallelization strategy used in Megatron-LM [50].



(a) Data/Sample. (b) Reduction. (c) Pipeline.

Figure 9: Representing different parallelization strategies for batched matrix multiplication with a PCG. s, i, o, and h indicate the sample, input channel, output channel, and attention head dimensions, respectively.

Substitution generation. To reduce the engineering effort to support new parallelization strategies, Unity automatically generates and formally verifies all valid PCG substitutions up to a fixed size to serve as a "basis set" from which the search algorithm can construct sophisticated optimizations. This also allows Unity to not only automatically discover algebraic transformations and parallelization strategies, but also to find novel hybrids of the two missed by prior approaches. To do so, Unity adopts TASO's super-optimization approach [25].

As in TASO, Unity discovers substitutions in two steps: first it uses a fast heuristic to identify candidate substitutions, and then it uses a more expensive formal verification to ensure correctness. To find candidate substitutions, Unity enumerates all possible PCGs up to a fixed size. Note that this fixed size does not limit the size of the transformations Unity can apply, as many larger substitutions are compositions of smaller ones.

For each generated PCG, Unity computes a *fingerprint*: a hash of the PCG's output tensors generated by evaluating the PCG on some fixed input tensors. To allow Unity to account for parallelization, we extend the fingerprint function in TASO [25] to include the parallelism degree of each tensor dimension. A pair of PCGs is considered a candidate substitution if both PCGs have an identical fingerprint. The addition of parallelism causes Unity to discover 651 new candidate substitutions beyond the 743 previously identified by TASO.

Substitution verification. Similar to TASO, Unity formally verifies the new substitutions using an automated theorem prover (Z3 [12] in our implementation). Operator specifications are provided in first-order logic, where an operator is represented as a function of its inputs and configuration parameters. For example, $\texttt{Reduce}(d,x)$ defines a `Reduce` operator with input x and parallelism degree d. The fact that `Reduce` commutes with matrix multiplication is captured by the following operator property (where $\texttt{Replicate}(d,y)$ represents a `Replicate` with input y and parallelism degree d):

$$\forall d,x,y.\ \texttt{Matmul}(\texttt{Reduce}(d,x),y) = \texttt{Reduce}(d,\texttt{Matmul}(x,\texttt{Replicate}(d,y)))$$

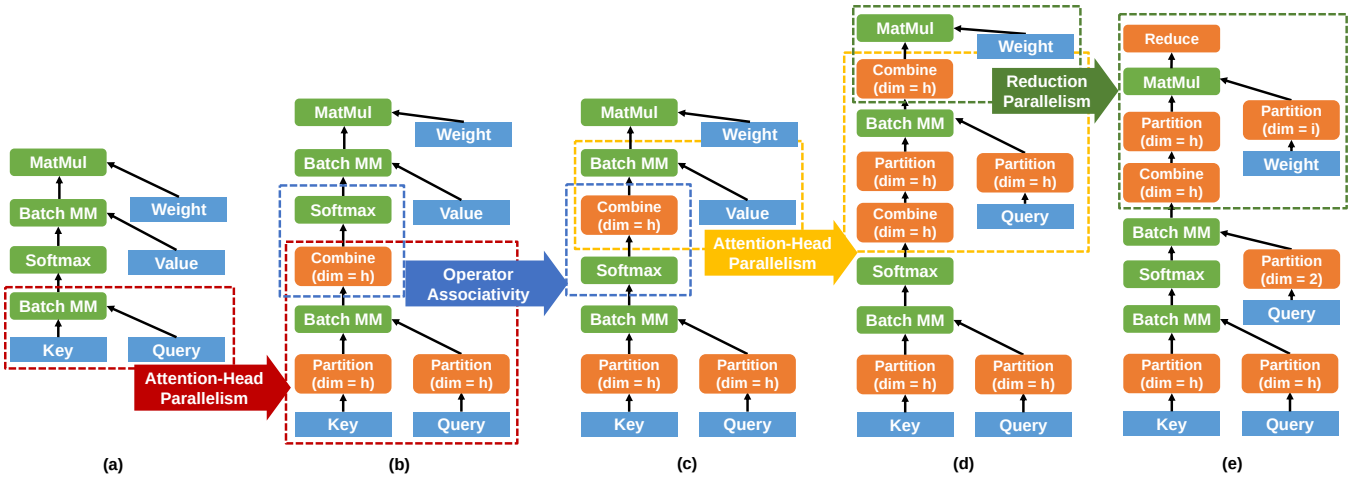


Figure 10: Representing the hand-tuned parallelization strategies used in Megatron-LM [50] as a sequence of basic graph substitutions in Unity. `BatchMM` and `MatMul` are batched and regular matrix multiplications, respectively. Each arrow denotes a graph substitution, where the dotted subgraphs in the same color are the source and target graph of the substitution. For `Partition` and `Combine`, the parentheses indicate the data dimension for which they are performed.

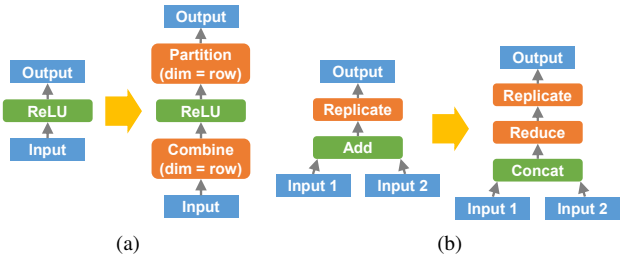


Figure 11: Substitution (a) shows that spatial parallelism is valid for `ReLU`. Substitution (b) demonstrates a hybrid algebraic-parallel transformation: transforming an `Add` into a `Concat` followed by a `Reduce` allows Unity to use the more efficient AllReduce communication pattern.

We follow TASO's methodology for developing operators' parallelization properties: we attempt to formally verify all candidate substitutions using Z3, and when a substitution cannot be verified but is correct, we add the missing operator properties. This procedure was repeated until all 651 new substitutions discovered by Unity were verified. Overall, we introduced 33 operator properties in addition to the 43 properties from TASO [25, Table 2] to verify all PCG substitutions.

Combined, the substitution generation and verification process takes a total of 30 minutes. Since the available substitutions only change on the addition of new operators or forms of parallelism, this process can be run entirely offline so as not to impact the execution time of Unity's joint search algorithm.

Example Substitutions. Most new substitutions generated by Unity simply state the parallelism valid for an operator. For instance, the substitution in Figure 11a indicates that `ReLU` supports spatial parallelism in the row dimension. However, combining algebraic transformations and parallelization also yields novel hybrids, such as the example shown in Figure 11b, where Unity identifies that an `Add` operator is equivalent to a `Concat` followed by a `Reduce`. In the left PCG, when `Input 1` and `Input 2` are located on separate devices and `Output` is required to be replicated across those same devices, `Input 1` and `Input 2` would have to be sent to and from a single device to be added. By applying this transformation, Unity is able to merge the input tensors into a single distributed tensor through a `Concat` (which moves no data) and replace the communication with a `Reduce` followed by a `Replicate` (which is implemented as an AllReduce).

5 Joint Optimization

This section describes Unity's search algorithm for jointly optimizing algebraic transformations and parallelization. The core problem is as follows: given a PCG (Section 3), a set of operator-level machine mappings (Section 3.2), and a set of PCG substitutions (Section 4), find (1) a sequence of PCG substitutions and (2) a machine mapping for the resulting PCG that minimize the per-iteration training time. A key challenge is the exponentially larger search space created by unifying algebraic transformations and parallelization. The search must also scale to both complex DNNs (i.e., a large input PCG) and large numbers of compute devices (i.e., a large set of operator-level machine mappings).

Unity uses a three-level hierarchical search algorithm, depicted in Figure 12. In simplified form, Unity breaks an input PCG into subgraphs, determines an optimized sequence of substitutions for each subgraph (which requires determining the optimized machine mapping for each candidate), and then

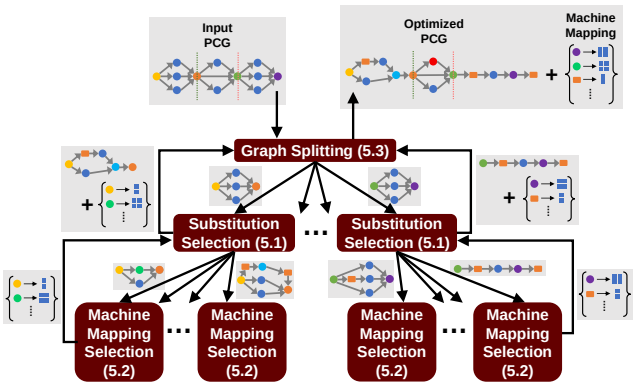


Figure 12: High-level depiction of Unity's hierarchical search.

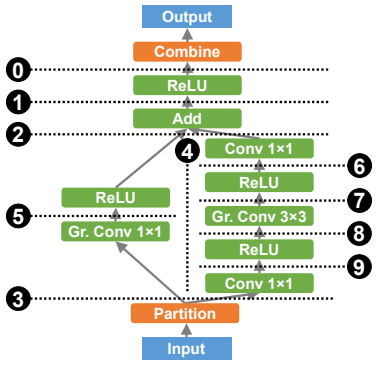


Figure 13: Applying sequence and parallel graph splits on a data-parallel ResNeXt module. Horizontal and vertical dotted lines refer to sequence and parallel splits, respectively, and numbers indicate the order they are applied.

combines these sub-solutions to produce the final output. This allows Unity to scale to DNNs with over 300 operators and machines with 192 GPUs while keeping search times below 20 minutes, which is negligible compared to the hours or days needed to train modern DNNs.

In the following section, we provide a more detailed description of Unity's search algorithm. Sections 5.1, 5.2, and 5.3 describe the three levels of Unity's search algorithm, starting from the middle layer (*substitution selection*), then the lowest (*machine mapping selection*), and finally introducing the highest level (*graph splitting*) as an optimization to help Unity scale to large DNNs. Afterward, we briefly address Unity's cost estimation and how the search algorithm can be tweaked to integrate pipeline parallelism.

5.1 Substitution Selection

Unity uses the cost-based backtracking search algorithm from TASO [25] to identify a sequence of substitutions that minimizes the execution time of an input PCG. Unity maintains a queue of candidate PCGs sorted by their execution times, and until the queue is emptied or a fixed budget is exceeded, Unity iteratively removes the best candidate from the queue and uses it to generate new candidates by applying every available substitution at every location in the PCG whenever applicable. Candidate PCGs with execution times that are a threshold factor times worse than the best candidate PCG seen so far are pruned, while the rest are inserted into the queue. The threshold factor allows the user to balance the search time and amount of exploration. In our experiments, we use a threshold factor of 1.05.[4]

This algorithm allows Unity to explore arbitrary sequences of substitutions, but requires an accurate cost estimator to evaluate the execution time of each candidate PCG. Since a PCG contains only the parallelization of each operator but not the devices to which it is assigned (i.e., the machine mapping), this cost estimator must first determine an optimized machine mapping. An efficient algorithm must be used to identify this mapping, as the cost estimator is called for every candidate PCG. Section 5.2 introduces our algorithm to find optimized machine mappings.

[4]This specific value was chosen to match [25].

5.2 Finding Optimized Machine Mappings

The lowest level of Unity's search algorithm identifies the optimized machine mapping for a candidate PCG. The key observation behind this level is that most modern DNN architectures consist of linear chains of independent strands of parallel computation. For example, ResNeXt [19] is built around two parallel strands of convolutions (see Figure 13), which are repeated to form the final model. Unity leverages this structure by recursively decomposing these linear chains and parallel strands into independent subgraphs through *sequence* and *parallel graph splits* respectively. Figure 13 demonstrates how sequence and parallel graph splits can be iteratively applied to decompose a ResNeXt module into recursive sub-problems which can be solved via dynamic programming.

A *sequence graph split* partitions an input PCG $\mathcal{G}$ by finding a postdominator node n, such that all paths from the inputs to the outputs of $\mathcal{G}$ go through n. This post-dominator node splits $\mathcal{G}$ into two disjoint subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$. Since all of $\mathcal{G}_2$ depends on n, and n depends on all of $\mathcal{G}_1$, every operator in $\mathcal{G}_1$ must complete before any in $\mathcal{G}_2$ can start. This reduces the task of finding an optimized machine mapping for $\mathcal{G}$ to optimizing machine mappings for $\mathcal{G}_1$, $\mathcal{G}_2$, and n. For example, for split ❶ in Figure 13, assuming no other splits (such as ❼) had already been applied, n would be the `Add` node, $\mathcal{G}_1$ would be all the nodes from `Input` up to but not including `Add`, and $\mathcal{G}_2$ would be all the nodes after the `Add` until `Output`.

A *parallel graph split* partitions a PCG $\mathcal{G}$ into independent subgraphs whose computations can be performed in parallel. In this case, Unity considers two potential ways of running the sides $\mathcal{G}_1$ and $\mathcal{G}_2$: in sequence (with access to the full machine resources) or in parallel (with each side given a disjoint share of the available resources) and chooses the faster one. Unity

does not allow combinations of serial and parallel execution, in which branches are run partially in parallel and partially in serial. While this eliminates certain strategies, considering them would significantly reduce Unity's scalability as it requires analyzing exponentially many interleavings of operators, and as evidenced by the results in Section 6, these strategies are not necessary to achieve good performance. To determine how to partition the available resources when running in parallel, Unity iterates over all possible resource quantities that can be assigned to each side. By considering resource quantities, Unity ignores redundant divisions that differ only in which GPUs are assigned and not in the number and location of these GPUs, replacing an exponential search over all subsets of devices with a quadratic search over resource quantities.

As an additional optimization, Unity maintains a cache of the selected machine mappings for all subgraphs. Since substitution selection generates a new candidate for each substitution, and each substitution modifies only a small part of a PCG, many candidate PCGs have most of their subgraphs in common with other candidates. This allows Unity to skip computing the cost and machine mapping of all but the part of the PCG modified by the substitution under consideration.

5.3 Scaling to Large Graphs

Even with the dynamic programming algorithm and cross-invocation caching, the search algorithm described so far fails to scale to large models. To understand why, we examine how the number of candidate PCGs in substitution selection scales with the size of the input PCG.

As described in Section 5.1, at each iteration Unity generates a candidate PCG for every possible application of each substitution. In the worst case this would require examining $O(2^{gs})$ candidates, where g is the number of nodes in the PCG and s is the number of substitutions Unity considers. In practice s has limited impact on search time as only a small fraction of the substitutions Unity considers can be applied to any one model, but for large models the exponential behavior of g becomes problematic.

To solve this, we borrow from Section 5.1 and decompose the PCG into independent sequential subgraphs. However, this approach prevents applying substitutions across these splits, which is problematic since Unity uses substitutions to represent parallelization. Thus, naive graph splitting would reduce the parallelism degree across all splits to 1, eliminating many common and important parallelization strategies, such as using data parallelism across the entire model.

Unity addresses this issue by explicitly searching for the optimal parallelization across every split location. More specifically, for every possible partitioning of the tensor communicated across the split, Unity optimizes the resulting two subgraphs under the condition that the first subgraph's output and the second subgraph's input must both match the partitioning under consideration. When either subgraph does not meet this condition, parallelization operators are inserted to ensure any communication cost arising from a change in partitioning is accounted for.

This method works for `Partition` and `Combine` but encounters a problem with `Replicate` and `Reduce`. For example, consider the case of the tensor crossing the split location having its replica degree fixed to 2 by the search algorithm. To coerce the first subgraph to output a tensor in this format, the search algorithm could insert a `Replicate` as its final operation, and the algorithm similarly could insert a `Reduce` as the first operation of the second subgraph. However, this will incorrectly scale the tensor by a factor of 2! The core issue is that unlike `Partition` and `Combine`, `Replicate` and `Reduce` are not inverses of each other. Fortunately, since reduction parallelism that spans many nodes of a computation graph is rarely useful in practice, we limit the partitionings across splits to only those with a replica degree of 1.

To reduce the number of algebraic transformations these splits prevent, Unity follows MetaFlow [26] and chooses split locations that disrupt the fewest substitutions while maintaining a minimum subgraph size k.[5] Thus graph splitting reduces the worst-case number of candidate PCGs from exponential in g to linear in g, specifically from $O(2^{gs})$ to $O\left(\frac{gp}{k} \times 2^{ks}\right)$ where p is the number of valid tensor partitionings.

Cost estimation. To estimate operator run times and communication costs we use similar methods as prior work [24, 27]. More accurate cost models are possible [47], but we have not noticed any issues caused by inaccuracies in our model.

Pipeline parallelism. When considering pipeline parallelism, Unity adopts the 1F1B schedule (i.e., interleaving forward and backward micro-batches on each device) and the weight update semantics from PipeDream-2BW [40], which achieves both high training throughput and low memory footprint. To reduce the search space, Unity only considers strategies where pipeline parallelism is applied to *all* operators in a PCG, since a non pipeline-parallel operator in the PCG would disable the benefits of pipeline parallelism. In addition, similar to prior work [20, 39, 65], Unity only considers sequential pipeline parallelism where each stage only communicates with a single next stage in the pipeline (except for the last stage, which directly performs back propagation after forward processing). Unity also follows prior work [16, 20, 53] in assuming that the number of micro-batches in a mini-batch is much larger then the number of pipeline stages so the additional latency introduced by pipeline initialization can be ignored. These constraints allow Unity to explore a comprehensive search space that includes existing pipeline parallelism strategies while maintaining reasonable search time. The search algorithm is also slightly modified: instead of using per-iteration run time as a proxy for throughput, we

[5] Our experiments use $k = 10$ as it strikes a balance between keeping subgraph sizes small enough for good scalability while blocking relatively few substitutions.

Task	Architecture	Dataset
Image Classification	ResNeXt-50 [60]	ImageNet [46]
	Inception-v3 [51]	ImageNet [46]
Language Models	BERT-Large [14]	WikiText-2 [35]
Recommendation Systems	DLRM [41]	Criteo Kaggle [4]
	XDL [28]	Criteo Kaggle [4]
Precision Medicine	CANDLE-Uno [3]	Dose response data [1]
Regression	MLP [17]	Synthetic data

Table 1: Overview of the seven DNNs evaluated.

maximize the throughput directly.

6 Evaluation

6.1 Implementation and Experimental Setup

Unity is implemented on top of FlexFlow [27], a distributed multi-GPU runtime for DNN training. We modified FlexFlow to represent models with PCGs, added support for Unity's additional forms of parallelism, and replaced FlexFlow's randomized search with the algorithm described in Section 5. The substitution generator (Section 4) is implemented on top of TASO [25], and extends its fingerprint function to consider parallelization. We also add 33 parallelization-specific properties that are used by the substitution verifier as axioms capturing the semantics of the parallelization operators.

All experiments were performed on the Summit supercomputer [2, 56]. Each compute node is equipped with two IBM POWER9 CPUs, 512 GB main memory, and six NVIDIA Volta V100 GPUs. Three of the GPUs within a node are connected to the same CPU and interconnected via NVLink. Nodes are connected with Mellanox EDR 100Gb InfiniBand.

DNNs. Table 1 summarizes the seven DNN models used in our evaluation. ResNeXt-50 [60] and Inception-v3 are commonly used DNNs for image classification. BERT [14] is a language model with state-of-the-art accuracy on a spectrum of language tasks. DLRM [41] and XDL [28] are deep learning recommendation models for personalization and ads recommendation. CANDLE-Uno [3] is a DNN architecture for precision medicine. Multi-layer perceptron [17] (MLP) is a widely used architecture for a variety of regression tasks and a core component in many DNNs.

We follow prior work in setting hyperparameters for training (e.g., batch sizes, learning rates) [3, 14, 38, 41, 60]. We report per-GPU minibatch size B: for runs with n GPUs, the global minibatch size is $n \times B$. The global minibatch sizes are consistent with those reported in the literature. We use a per-GPU minibatch size of 64 for ResNeXt-50 and Inception-v3, 4 for BERT-Large, 1024 for DLRM and XDL, and 256 for CANDLE-Uno and MLP. The MLP model includes 16 dense layers, each of which has a hidden dimension of 8192. We use Adam [29] with a learning rate of 0.0001 for BERT-Large, and SGD [18] with a learning rate of 0.01 for the other DNNs.

Unless stated, pipeline parallelism is disabled when comparing against frameworks that do not support this feature. We evaluate the impact of pipeline parallelism in Figure 15a.

Search Time. For all DNNs except Inception-v3, Unity's search times are under 10 minutes even for the largest GPU count (i.e., 192). Even for Inception-v3, the most complex architecture in our evaluation with 323 operators, search terminates within 20 minutes. These times are negligible compared to the hours or days needed to train these DNNs.

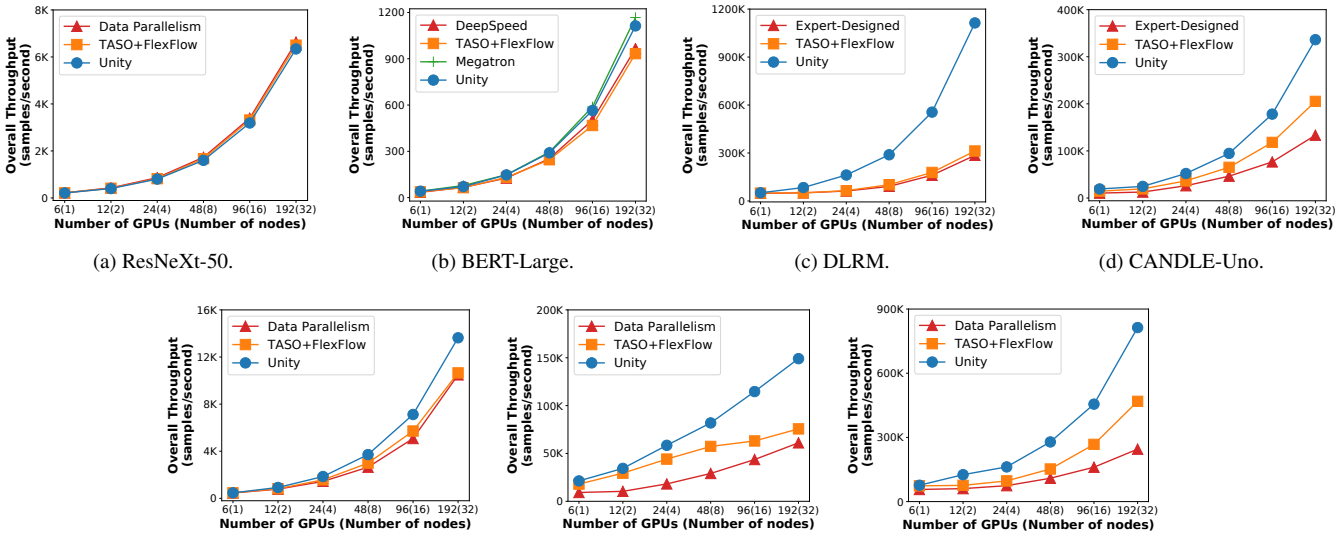
6.2 End-to-end Evaluation

We compare the end-to-end training performance of Unity and existing frameworks such as Megatron [50] and DeepSpeed [43]. We also compare against using TASO [25] and FlexFlow [27] to perform sequential optimization (i.e., TASO first and FlexFlow second). Since Megatron [50] and DeepSpeed [43] require the user to manually optimize each model, these baselines are only present for a subset of the models, while the automated approaches of FlexFlow and Unity can be used across all seven. Figure 14 shows the results.

BERT-Large has been highly optimized by existing frameworks such as Megatron and DeepSpeed which use expert-designed strategies combining multiple forms of parallelism. As such, Unity is not expected to outperform these strategies. Instead, the primary purpose of this evaluation is to determine if Unity can re-discover these hand-tuned strategies within a few minutes of automated search. Note that since Megatron and DeepSpeed require users to manually specify all parallelism degrees for data, tensor-model, and pipeline parallelism, we explore different combinations of the supported parallelism degrees and report the best performance.

Unity achieves on-par performance with Megatron and outperforms both DeepSpeed and FlexFlow. We find that the best strategy discovered by Unity is almost the same as the expert-designed strategy in Megatron: the only difference is that for some matrix multiplications Megatron uses reduction parallelism while Unity uses data parallelism, which has a negligible impact on overall training performance. This shows that even on highly-optimized models Unity is able to automatically generate parallelization optimizations that match those manually designed by domain experts. Megatron is customized for Transformer-based language models and does not support the other DNNs in our evaluation. The fact that the parallelization strategy discovered by Unity matches the expert-designed strategy in Megatron is, in our view, a positive outcome of Unity.

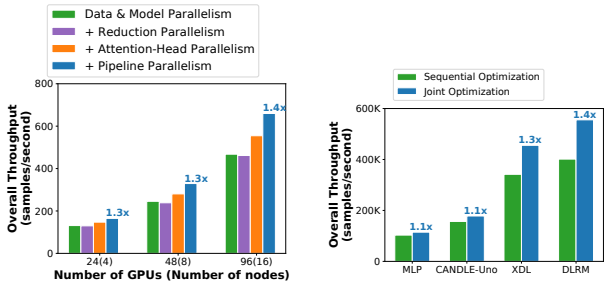
DLRM and CANDLE-Uno both exceed the memory capacity of a single GPU, preventing data parallel training. For both models we use the expert-designed strategy proposed in [38] as a baseline, which parallelizes communication-intensive operators (e.g., embedding tables) in model parallelism and compute-intensive operators (e.g., matrix multiplications) in data parallelism. Unity outperforms both expert-designed strategies and TASO+FlexFlow by up to 3.6× on DLRM and 1.6× on CANDLE-Uno. For all other models, we compare Unity against data parallelism and TASO+FlexFlow.



(a) ResNeXt-50. (b) BERT-Large. (c) DLRM. (d) CANDLE-Uno.

(e) Inception-v3. (f) MLP. (g) XDL.

Figure 14: Training throughput comparison among existing frameworks and Unity. The experiments were performed on the Summit supercomputer [2] with 6 GPUs per node. All numbers were measured by averaging 1,000 training iterations.



(a) (b)

Figure 15: (a) End to end performance of BERT-Large integrating different parallelization dimensions. Speedups relative to data+model parallelism. (b) Speedups solely attributable to joint vs sequential optimization on 96 V100 GPUs (16 nodes). Search space and algorithm are fixed to remove effects from Unity's larger search space and improved search scalability.

Unity outperforms the best existing approaches by 1.0× on ResNeXt-50, 1.3× on Inception-v3, 2.0× on MLP, and 1.9× on XDL. The lack of improvement on ResNeXt-50 is expected as the model's optimal strategy (data parallelism) is already the default used by most frameworks.

We observe that the performance improvement is achieved by (1) supporting operator-specific parallelism and (2) jointly optimizing algebraic transformations and parallelization. We further analyze these details in the following experiments.

6.3 Parallelism Dimensions

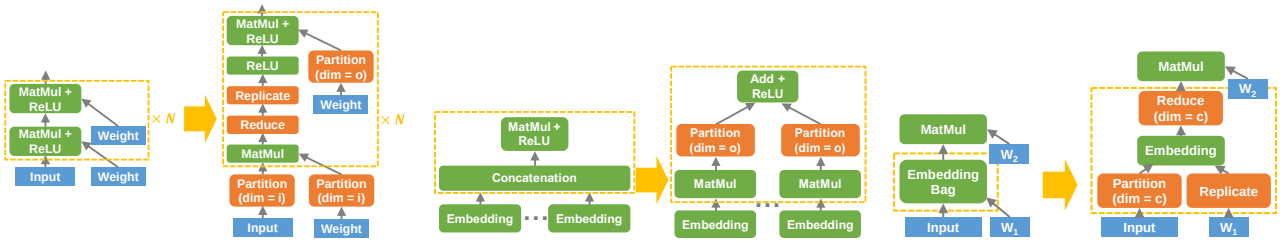
To evaluate how different parallelism dimensions improve training performance, we perform an ablation study of Unity on BERT-Large by iteratively adding new dimensions to Unity and measuring the training throughput. Figure 15a shows the results. Compared to data and model parallelism, adding reduction parallelism does not improve training performance, but combining reduction and attention-head parallelism increases performance by up to 1.2× because optimizing the attention operators in BERT-Large requires both reduction and attention-head parallelism, as shown in Figure 10. Enabling pipeline parallelism achieves an overall speedup of 1.4×. This result shows that hybrid strategies and operator-specific dimensions are critical for DNN training performance.

6.4 Joint Optimization

To evaluate Unity's joint optimization, we compare against sequential optimization of algebraic transformations and parallelization. Results are shown in Figure 15b. Unlike the TASO+FlexFlow baseline in Figure 14, in Figure 15b we include Unity's additional parallelism dimensions and improved scalability to isolate the effects of joint optimization. As a result, the performance improvement (up to 1.4× speedup) comes solely from the ability to optimize jointly rather than sequentially. We study three examples in detail.

The first (Figure 16a) is a slight generalization of the example introduced in Figure 2. By not fusing the first `MatMul` and `ReLU`, which would be done in sequential optimization as the algebraic optimizer would ignore parallelism, Unity is able to significantly reduce the amount of communication by using reduction parallelism and a more efficient AllReduce (represented by the `Reduce` followed by `Replicate`).

The second is shown in Figure 16b. `Concatenation` is the main performance bottleneck in DLRM and XDL, since it can-



(a) Multi-layer perceptron. (b) `Concatenation`. (c) `EmbeddingBag`.

Figure 16: Example joint optimizations of computation graph and parallelization discovered by Unity. For `Partition`, *i* and *o* indicate the input and output channel dimensions of a matrix multiplication.

Table 2: Search algorithm ablation study. "Scaled" numbers are relative to the 2 GPU time with all optimizations enabled.

	All		w/o Split		w/o Cache+Split	
	Time	Scaled	Time	Scaled	Time	Scaled
6 GPUs (1 nodes)	57s	1×	4m 01s	4.3×	37m 01s	38.5×
12 GPUs (2 nodes)	1m 47s	1.9×	11m 15s	16.8×	> 1h	n/a
24 GPUs (4 nodes)	3m 00s	3.1×	> 1h	n/a	> 1h	n/a
48 GPUs (8 nodes)	5m 55s	6.1×	> 1h	n/a	> 1h	n/a

not be parallelized in the same dimension as the `Embedding` operators and requires an all-to-all synchronization. The optimization eliminates the `Concatenation` by replacing the subsequent `MatMul` with independent `MatMul`s executed using the same model parallel strategy as the `Embedding` operators, which reduces communication costs as the `Embedding` operators' outputs are only used locally.

The third optimization is shown in Figure 16c. An `EmbeddingBag` [15] operator computes the sum of a bag of embeddings for each training sample. Unity discovers a joint optimization that transforms an `EmbeddingBag` to a normal `Embedding` to enable additional parallelization opportunities.

6.5 Search Algorithm

To evaluate the impact of the three search optimizations (graph splitting, cross-invocation cache, and dynamic programming) presented in Section 5, we perform an ablation study of the search time for ResNeXt-50. With all three techniques enabled (the "All" column), we see roughly linear scaling as we move from 6 to 48 GPUs. This, along with Figure 14, demonstrates that Unity's search algorithm scales to nontrivial node counts.

Disabling graph splitting increases search times by 4.3-8.8× and causes them to scale nonlinearly, while disabling the cross-invocation cache adds an additional 8.9×. Disabling the dynamic programming algorithm causes even the smallest cases to time out. These results indicate that the three proposed techniques are necessary for adequate performance.

7 Related Work

Manually-designed parallelization strategies. Manually-designed parallelization strategies are used in most existing DNN frameworks to optimize distributed DNN training [5, 6, 42, 43, 49]. For example, Neo [38] optimizes DLRM by using data parallelism for compute-intensive operators and model parallelism for communication-intensive operators. Megatron-LM [50] proposes a model-specific customized strategy that combines data, reduction, and attention-head parallelism for training large language models. These strategies only work for specific DNN models and do not generalize. We use these expert-designed strategies as baselines in our evaluation and show that Unity can automatically discover strategies with improved performance.

Automated DNN parallelization. Recent work has proposed automated approaches to optimizing distributed DNN training. For example, ColocRL [36, 37] and Placeto [7] use reinforcement learning to find efficient device placement for model parallelism. Baechi [22] achieves fast device placement for model parallelism using two memory-constrained algorithms. FlexFlow [27] uses randomized search to optimize data, model, and spatial parallelism. GSPMD [61], a generalization of GShard [34], finds parallelization strategies based on user-provided hints. PipeDream [39] uses dynamic programming to find optimized strategies combining pipeline and data parallelism. Tofu [59] uses recursive search to minimize communication time and automatically discovers parallelization dimensions via interval analysis. Tarnawski et al. [52, 53] propose a two-level dynamic programming algorithm to partition a DNN computation graph across devices by combining data, pipeline, and tensor model parallelism. Alpa [65] automates inter-operator (i.e., pipeline) parallelism using dynamic programming and intra-operator (i.e., data and tensor model) parallelism using integer linear programming. Whale [23] uses computation-balanced partitioning to accomodate heterogenous compute devices and allows specifying parallelization strategies through small parallelization primitives. TensorOpt [8] introduces the *cost frontier* to simultaneously reason about multiple objectives (e.g., execution time and cloud resource cost) in automatic parallelization. Finally,

AutoSync [63] learns to optimize synchronization strategies for data-parallel training from a few thousand samples. However, existing approaches (except Tofu) only support limited parallelism dimensions and none jointly optimizes algebraic transformations and parallelization. Unity supports all existing parallelism dimensions, is extensible to new operators and forms of parallelism, and jointly optimizes algebraic transformations and parallelization.

Automated algebraic transformations. TASO [25] automatically discovers algebraic transformations for DNNs but does not support parallelization. Unity adopts the superoptimization idea from TASO to generate and verify PCG substitutions. However, unlike the algebraic transformation task considered by TASO, Unity deals with a significantly larger search space and considers additional tasks, such as device assignments. We observe that TASO's search algorithm alone is incapable of exploring the larger search space. To address this challenge, Unity introduces three novel elements of the search technique: the dynamic programming algorithm for finding optimized machine mappings, the subgraph cache for exploiting the locality of graph substitutions, and the additional parallelism-compatible divide-and-conquer approach to enabling scalability to complex models.

Automated DNN code generation. Recent work has proposed approaches for generating hardware-specific code for DNN operators. TVM [10, 11] uses a learning-based algorithm to generate optimized code for a diverse set of hardware backends. Ansor [64] extends TVM by utilizing a hierarchical search algorithm to explore a much larger search space of program candidates. Unity optimizes DNN computation at a higher level than these approaches. Therefore, Unity's optimizations are orthogonal and can be combined with existing code generation techniques. We leave integrating code generation into Unity as future work.

Intermediate representations for DNN parallelization. TensorFlow [**?**], MLIR [31, 32], Relay [45], and ONNX [33] represent DNN computation with graph-based intermediate representations (IRs). Distributed training of a model is represented by annotating each operator with a parallelization strategy describing how the operator is parallelized across devices. These approaches represent algebraic transformations and parallelization separately and optimize them sequentially, missing joint optimizations. pONNX [57], automap [48], and DistIR [47] propose IRs that express both computation and communication, but are too low-level to be used for Unity-style joint optimization (see Section 3.4 for details). Unity uses a higher-level representation better suited to optimization, the PCG, and represents both parallelization and algebraic transformations as graph substitutions on PCGs.

8 Limitations and Future Work

To scale to large DNNs and machines, Unity's search algorithm exploits the sequential and parallel structure of modern DNNs (see Section 5.2). However, there exist DNN architectures (e.g., NASNet [66]) that violate this structure. Extending Unity to include these DNNs would improve generality, but potentially at the cost of decreased scalability.

While Unity successfully optimizes two of the most prominent classes of optimizations (i.e., algebraic transformations and parallelization), there are a variety of additional optimizations currently not considered, such as tensor offloading and rematerialization [21, 30, 44]. The PCG can be extended to represent these optimizations, but the search algorithm as presented in Section 5 does not reason about memory usage and therefore may generate parallelization strategies that violate memory constraints. While these invalid strategies can be made valid by applying the necessary tensor offloading and rematerialization afterward, not including these optimizations in Unity's joint search potentially leads to suboptimal performance. Thus, integrating memory optimizations into Unity's search algorithm is a promising area for future research.

Another limitation of Unity is its support for pipeline parallelism. While PCGs are capable of representing parallelization strategies that interleave pipeline-parallel and non-pipeline-parallel operators in a PCG, our search algorithm excludes these cases to reduce the search space. In addition, Unity's search algorithm does not consider non-sequential pipeline parallelism strategies, where a stage can have multiple predecessor/successor stages.

9 Conclusion

This paper presents Unity, the first system that jointly optimizes algebraic transformations and parallelization in distributed DNN training. Unity represents both parallelization and algebraic transformations as substitutions on a unified graph representation, uses a novel hierarchical search algorithm to identify an optimized sequence of substitutions, and scales to large numbers of GPUs and complex DNNs.

Our evaluation with seven real-world DNN benchmarks on up to 192 GPUs show that Unity outperforms state-of-the-art parallelization approaches by up to 3.6× while keeping optimization times under 20 minutes. As nearly half of this speedup is attributable solely to the use of joint optimization over sequential optimization, Unity demonstrates that joint optimization is practical and that future systems will need to include it or else miss significant performance gains.

Acknowledgement

We thank the anonymous reviewers for their comments, and are grateful to our shepherd Byung-Gon Chun for his feedback. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656518, and an NSF award CNS-2147909. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] CANDLE Benchmarks. https://github.com/ECP-CANDLE/Benchmarks, 2018. 3, 11

[2] Summit supercomputer. https://www.olcf.ornl.gov/summit/, 2018. 11, 12

[3] Uno: Predicting tumor dose response across multiple data sources. https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/Uno, 2018. 11

[4] Criteo 1tb click logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/, 2021. 11

[5] Optimize and accelerate machine learning inferencing and training. https://www.onnxruntime.ai/, 2021. 13

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016. 1, 3, 13

[7] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *CoRR*, abs/1906.08879, 2019. 13

[8] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. TensorOpt: Exploring the Tradeoffs in Distributed DNN Training with Auto-Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2021. 13

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. 3

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. 14

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*, NeurIPS'18. 2018. 14

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008. 7

[13] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012. 1

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. 3, 11

[15] EmbeddingBag in PyTorch. https://pytorch.org/docs/stable/generated/torch.nn.EmbeddingBag.html, 2021. 13

[16] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 431–445, New York, NY, USA, 2021. Association for Computing Machinery. 10

[17] Matt W Gardner and SR Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15):2627–2636, 1998. 11

[18] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017. 11

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016. 9

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2018. 10

[21] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511, 2020. 14

[22] Beomyeol Jeon, Linda Cai, Pallavi Srivastava, Jintao Jiang, Xiaolan Ke, Yitao Meng, Cong Xie, and Indranil Gupta. Baechi: Fast device placement of machine learning graphs. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 416–430, New York, NY, USA, 2020. Association for Computing Machinery. 13

[23] Xianyan Jia, Le Jiang, Ang Wang, Jie Zhang, Xinyuan Li, Wencong Xiao, Langshi chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Scaling Deep Learning Model Training to the Trillions. *arXiv:2011.09208 [cs]*, August 2021. 2, 13

[24] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*. PMLR, 2018. 1, 10

[25] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery. 1, 3, 4, 7, 8, 9, 11, 14

[26] Zhihao Jia, James Thomas, Todd Warzawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019. 1, 10

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019. 1, 2, 3, 10, 11, 13

[28] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, Liqin Zhao, Zhi Wang, Peng Sun, Yu Zhang, Di Zhang, Jinhui Li, Jian Xu, Xiaoqiang Zhu, and Kun Gai. Xdl: An industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, DLP-KDD '19, New York, NY, USA, 2019. Association for Computing Machinery. 11

[29] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 11

[30] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. *CoRR*, abs/2006.09616, 2020. 14

[31] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. 14

[32] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020. 14

[33] Tung D. Le, Gheorghe-Teodor Bercea, Tong Chen, Alexandre E. Eichenberger, Haruki Imai, Tian Jin, Kiyokuni Kawachiya, Yasushi Negishi, and Kevin O'Brien. Compiling ONNX neural network models using MLIR. *CoRR*, abs/2008.08272, 2020. 14

[34] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020. 13

[35] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016. 11

[36] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018. 13

[37] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. 2017. 13

[38] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong

Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2021. 11, 13

[39] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery. 1, 4, 10, 13

[40] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training, 2020. 10

[41] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019. 3, 11

[42] Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017. 3, 13

[43] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training A trillion parameter models. *CoRR*, abs/1910.02054, 2019. 11, 13

[44] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021. 14

[45] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level IR for deep learning. *CoRR*, abs/1904.08368, 2019. 14

[46] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 2015. 11

[47] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Tim Harris, and Matei Zaharia. DistIR: An Intermediate Representation and Simulator for Efficient Neural Network Distribution. *arXiv:2111.05426 [cs]*, November 2021. 7, 10, 14

[48] Michael Schaarschmidt, Dominik Grewe, Dimitrios Vytiniotis, Adam Paszke, Georg Stefan Schmid, Tamara Norman, James Molloy, Jonathan Godwin, Norman Alexander Rink, Vinod Nair, and Dan Belov. Automap: Towards Ergonomic Automated Parallelism for ML Models. *arXiv:2112.02958 [cs]*, December 2021. 2, 7, 14

[49] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. *arXiv:1811.02084 [cs, stat]*, November 2018. 13

[50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. 1, 7, 8, 11, 13

[51] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016. 11

[52] Jakub Tarnawski, Amar Phanishayee, Nikhil R. Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators, 2020. 13

[53] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 24829–24840. Curran Associates, Inc., 2021. 10, 13

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. 4

[55] Sudharshan S Vazhkudai, Bronis R de Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle,

Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC. IEEE, 2018. 5

[56] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018. 11

[57] Fei Wang, Guoyang Chen, Weifeng Zhang, and Tiark Rompf. Parallel Training via Computation Graph Transformation. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3430–3439, December 2019. 3, 7, 14

[58] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021. 1

[59] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–17, New York, NY, USA, March 2019. Association for Computing Machinery. 1, 2, 3, 13

[60] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016. 11

[61] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. GSPMD: General and Scalable Parallelization for ML Computation Graphs. *arXiv:2105.04663 [cs]*, May 2021. 13

[62] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality Saturation for Tensor Graph Superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, March 2021. 2

[63] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. Autosync: Learning to synchronize for data-parallel distributed deep learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 906–917. Curran Associates, Inc., 2020. 14

[64] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor : Generating high-performance tensor programs for deep learning. *CoRR*, abs/2006.06762, 2020. 14

[65] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. *CoRR*, abs/2201.12023, 2022. 10, 13

[66] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018. 14

Trinity: High-Performance Mobile Emulation through Graphics Projection

Di Gao[†*], Hao Lin[†*], Zhenhua Li[†], Chengen Huang[†], Yunhao Liu[†]
Feng Qian[§], Liangyi Gong[‡], Tianyin Xu[¶]
[†]*Tsinghua University* [§]*University of Minnesota* [‡]*CNIC, CAS* [¶]*UIUC*

Abstract

Mobile emulation, which creates full-fledged software mobile devices on a physical PC/server, is pivotal to the mobile ecosystem, especially for PC-based mobile gaming, app debugging, and malware detection. Unfortunately, existing mobile emulators perform poorly on graphics-intensive apps in terms of either efficiency or compatibility or both. To address this, we introduce *graphics projection*, a novel graphics virtualization mechanism that adds a small-size *projection space* inside the guest memory of a virtual mobile device. The projection space processes graphics operations involving control contexts and resource handles without host interactions. Novel flow control and data teleporting mechanisms are devised to match the decoupled graphics processing rates of the virtual device and the host GPU to maximize performance. The resulting new Android emulator, dubbed Trinity, exhibits an average of 93.3% native hardware performance and 97.2% app support, in some cases outperforming other emulators by more than an order of magnitude. It has been adopted by Huawei DevEco Studio, a major Android IDE with millions of developers.

1 Introduction

Mobile emulation has been a keystone of the mobile ecosystem. Developers today typically debug their apps on generic mobile emulators (*e.g.,* Google's Android Emulator, or GAE for short) rather than on heterogeneous real devices. Also, various dedicated mobile emulators (*e.g.,* Bluestacks [14] and DAOW [55]) are used to detect malware in app markets [21, 44, 54], to enable mobile gaming on PCs [14, 55], and to empower the emerging notion of cloud gaming [36].

1.1 Motivation

To create full-fledged software mobile devices on a physical PC/server, mobile emulators usually adopt the classic virtualization framework [33, 40, 45, 46] where a mobile OS runs in a virtual machine (VM), referred to as the guest, hosted on a PC/server, referred to as the host. However, traditional virtualization techniques are initially designed to work on headless servers or common PCs without requiring strong UI interactions within the VM, while real-world mobile apps are highly interactive [37] and thus expecting mobile emulators to have powerful graphics processing capabilities (as provided by real mobile phones) [55]. This *capability gap* is further aggravated by the substantial architectural differences between the graphics stacks of desktop and mobile OSes [15].

Over the years, several approaches have been proposed to fill the gap. Perhaps the most intuitive is solely relying on a CPU to carry out a GPU's functions. For example, as a user-space library residing in mobile OSes (*e.g.,* Android), SwiftShader [26] helps a CPU mimic the processing routines of a GPU. This achieves the best compatibility since any mobile app can thus seamlessly run under a wide variety of environments even without actual graphics hardware, but at the cost of poor efficiency since a CPU is never suited to handling the highly parallel (graphics) rendering tasks.

To improve the emulation efficiency, a natural approach is multiplexing the host GPU within a PC/server through API remoting [18, 50], which intercepts high-level graphics API calls at the guest and then executes them on the host GPU with dedicated RPC protocols and guest-host I/O pipes. Unfortunately, the resulting products (*e.g.,* GAE) cannot smoothly run many common apps, let alone "heavy" (*i.e.,* graphics-intensive) apps for AR/VR viewing and 3D gaming. This shortcoming stems from frequent VM Exits to the host to execute API calls, introducing a considerable "tromboning" effect [19] on the control and data flows. This results in additional idle waiting at the guest, as it must wait not only for the API call to complete, but also for the added process of exiting to the host and returning back to the guest.

To mitigate the issue, *device emulation* [17] moves the virtualization boundary from the API level to the driver level. It forwards guest-side graphics driver commands to the host with a shared memory region inside the guest kernel to realize their effects with the host GPU. Compared to high-level APIs, driver commands are much fewer, more capable, and mostly asynchronous [17], so device emulation effectively reduces guest-host control/data exchanges and idle waiting. However, the translation from API calls to driver commands degrades critical high-level abstractions such as windows and threads to low-level memory addresses and register values. Due to the loss of high-level information, driver commands must be sequentially executed at the host, degrading guest-side multi-threaded rendering to host-side single-threaded rendering. Hence, the resulting emulators (*e.g.,* QEMU-KVM) can smoothly run regular apps but not heavy ones.

[*] Co-primary authors. Zhenhua Li is the corresponding author.

Another approach is to break guest-host isolation by removing the virtualization layer so apps can directly use the GPU, as embodied in DAOW [55]. This requires manually translating Linux system calls used by Android to Windows ones. Unfortunately, many apps cannot run on DAOW because many (∼46%) system calls are not translated due to the huge engineering efforts required for full system calls' translation. Also, the supported apps must run under the protection of additional sophisticated security defenses to compensate for the lack of guest-host isolation.

1.2 Contribution

We present Trinity, a novel mobile emulator that simultaneously achieves high efficiency and compatibility. Our guiding principle is to decouple the guest-host control and data exchanges and make them as asynchronous as possible when multiplexing the host GPU under the virtualization framework, so that frequent VM Exits for synchronous host-side execution of API calls can be largely reduced. For this purpose, we propose to add a *projection space* inside the guest memory, where we selectively maintain a "projected" subset of control contexts (termed *shadow contexts*) and resource handles. Such contexts and handles are derived but different from the real ones required by a physical GPU to perform rendering, so as to reflect and reproduce the effects of guest-side graphics operations (*i.e.,* API calls). Thus, the vast majority (99.93%) of graphics API calls do not need synchronous execution at the host, while consuming less than 1 MB memory for even a heavy 3D app.

Concretely, when an Android app wants to draw a triangle on a physical phone, it sequentially issues three types of graphics API calls: context setting (Type-1), resource management (Type-2), and drawing (Type-3). Type-1 prepare the canvas and bind resource handles; Type-2 populate the handles' underlying resources with the triangle's vertex coordinates, filling colors/patterns, *etc.*; Type-3 instruct the GPU to render and display the triangle. In contrast, as shown in Figure 1, when the app runs in Trinity, Type-1 and Type-2 calls are first executed only in the projection space, *i.e.,* their effects are temporarily reflected on the shadow contexts and resource handles. Later upon drawing calls (Type-3), their effects are delivered to the host to realize actual rendering.

Combined with graphics projection, an elastic flow control algorithm is devised in Trinity to orchestrate the execution speeds of control flows at both the guest and host sides. Regarding the guest-host data flows, we find that the major challenge of rapidly delivering them lies in the high dynamics of system status and data volume (*e.g.,* bursty data flows are common in graphics operations). To this end, we find that the dynamic situations in fact follow only a few patterns, each of which requires specific data aggregation, persistence, and arrival notification strategies. Therefore, we implement all the required strategies, and utilize *static timing analysis* [12]

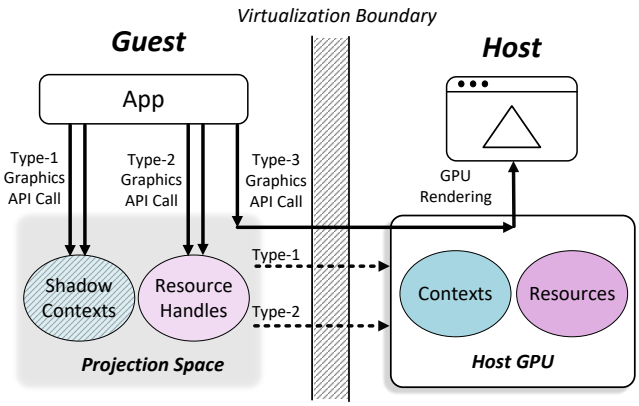


Figure 1: Basic workflow of Trinity.

to estimate which strategy is best suited to a data flow. With these efforts, we achieve high emulation efficiency for Trinity.

Similar to GAE, Trinity is also implemented atop QEMU (for general device extensibility) and hosts the Android OS, with 118K lines of C/C++ code. We evaluate its performance using standard graphics benchmarks, the top-100 3D apps from Google Play, and 10K apps randomly selected from Google Play. We also compare the results with six mainstream emulators: GAE, QEMU-KVM, Windows Subsystem for Android, VMware Workstation, Bluestacks, and DAOW. The evaluation shows that Trinity can achieve 80%∼110% (averaging at 93.3%) native hardware performance, outperforming the other emulators by 1.4× to 20×. For compatibility, Trinity can run the top-100 3D apps and 97.2% of the 10K randomly selected apps. To our knowledge, Trinity is the first and the only Android emulator that can smoothly run heavy 3D apps without losing compatibility (or security).

Software/Code/Data Availability. Trinity has recently been adopted by Huawei DevEco Studio [28], a major Android IDE (integrated development environment) with millions of developers. Currently, it is going through the beta test run for minor functional adjustments and bug fixes. The binary, code, and measurement data involved in this work are released at `https://TrinityEmulator.github.io/`.

2 Understanding Mobile Graphics APIs

We first delve into the three types of APIs in OpenGL ES, the de facto graphics framework of Android (§2.1), and then measure real-world 3D apps to obtain an in-depth understanding of their graphics workloads (§2.2).

2.1 Background

Figure 2 shows a basic OpenGL ES program for drawing a triangle. The program creates a graphics buffer in a GPU's graphics memory using a Type-2 API—`glGenBuffers`, populates the buffer with the coordinate data of the triangle's

```
float vertices[9] = {  0.0f,  0.5f,  0.0f, // First vertex
                      -0.5f, -0.5f,  0.0f, // Second vertex
                       0.5f, -0.5f,  0.0f  // Third vertex
};  // Triangle vertices' (x, y, z) coordinates

float *vtx_mapped_buf; // Address of the mapped buffer

void populate_buffer() {
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
                                0, GL_DYNAMIC_DRAW);
    ...
    // Type-2: query the buffer size
    int buf_size;
    glGetBufferParameteriv(GL_ARRAY_BUFFER,
                GL_BUFFER_SIZE, &buf_size);
    // Type-2: map the buffer to main memory space
    vtx_mapped_buf = glMapBufferRange(GL_ARRAY_BUFFER,
                      0, buf_size, GL_MAP_WRITE_BIT);

    memcpy(vtx_mapped_buf, vertices, buf_size);
    // Type-2: unmap the buffer
    glUnmapBuffer(GL_ARRAY_BUFFER);
}
```

(a) Populate the bound graphics buffer by latent mapping.

```
uint vertex_buffer_handle; // Graphics buffer handle

void draw() {
    ...
    // Type-2: allocate a buffer and generate its handle
    glGenBuffers(1, &vertex_buffer_handle);

                1. The buffer's handle is bound to the context
    // Type-1: bind the buffer to context
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_handle);

    populate_buffer();
    ...
    // Type-3: draw the triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);
    ...
}
```

2. All subsequent operations do not need to specify the handle again

(b) Draw the triangle.

Figure 2: OpenGL ES code snippet for drawing a triangle.

vertices through a Type-1 API—`glBindBuffer` and a Type-2 API—`glMapBufferRange`, and then instructs the GPU to draw the triangle using a Type-3 API—`glDrawArrays`.

Type-1: Context Setting. To manipulate or use the allocated graphics buffer, instead of passing the buffer's handle to every API call, the program first calls `glBindBuffer`, which binds the handle to a thread-local *context*, *i.e.,* the transparent, global state of the thread. Then, all the subsequent buffer-related API calls (*e.g.,* the buffer population call `glBufferData` and the drawing call `glDrawArrays` that uses the buffer data to draw) will be directly applied to the bound buffer, without needing to specify the buffer handle in their call parameters.

The above process is called *context setting*, which configures critical information of the current thread's context. This programming paradigm avoids repeatedly transferring context information from the main memory to the GPU, particularly when the information is rarely modified. In general, the context information that requires setup includes the current *operation target*, *render configurations*, and *resource attributes*. The operation target identifies the object that subsequent API calls will affect, *e.g.,* in Figure 2 the buffer handle becomes the operation target of subsequent API calls after it is bound to the context. Render configurations define certain rendering behaviors, *e.g.,* whether to perform validation of pixel values after a frame is rendered. Resource attributes correspond to resources' internal information, *e.g.,* formats of images and data alignment specifications.

Type-2: Resource Management. Resources involved in graphics rendering include *graphics buffers* that store vertice and texture data (*"what to draw"*), *shader programs* that produce special graphics effects such as geometrical transformation (*"how to draw"*), and *sync objects* that set time-wise sync points (*"when to draw"*). Graphics buffers hold most of the graphics data and thus require careful management. To populate a buffer with graphics data, there are mainly two approaches—*immediate copy* and *latent mapping*.

With regard to immediate copy, data are passed into the `glBufferData` API's third call parameter and copied from the main memory to the bound graphics buffer, *i.e.,* the buffer underlying `vertex_buffer_handle`. This approach is easy to implement but involves synchronous, time-consuming memory copies. In contrast, Figure 2 shows the latent mapping approach, where `glBufferData` is called but no data are passed to it; `glMapBufferRange` instead maps the graphics buffer to a main memory address, *i.e.,* `vtx_mapped_buf`. The data can then be directly stored in the mapped main memory space, without needing to synchronously trigger memory-to-GPU copies. The data are latently copied to the graphics buffer by the GPU's hardware *copy engine* (a DMA device) usually when `glUnmapBuffer` is called to release the address mapping, thus being more flexible and efficient.

Type-3: Drawing. After the contexts and resources are prepared, the drawing phase is usually realized with just a few API calls, *e.g.,* `glDrawArrays` as shown in Figure 2. Such APIs are all designed to be asynchronous in the first place, so that the graphics processing throughput of a hardware GPU can be maximized. When a drawing call is issued, the call is simply pushed into the GPU's command queue rather than being executed synchronously.

Apart from the above operations for rendering a single frame, graphics apps often need to render continuous frames (*i.e.,* animations) in practice. To this end, a modern graphics app usually follows the *delta timing* principle [16] of graphics programming, where the app measures the rendering time of the current frame (referred to as the frame's delta time) to decide which scene should be rendered next. For example, when a game app renders the movement of a game character, the app would measure the delta time of the current frame to compute how far the character should move (*i.e.,* the character's coordinate change) in the next frame based on the delta time and the character's moving speed.

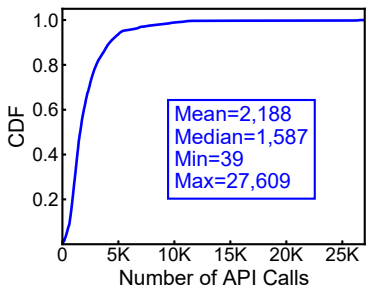


Figure 3: Number of API calls issued for rendering a single frame.

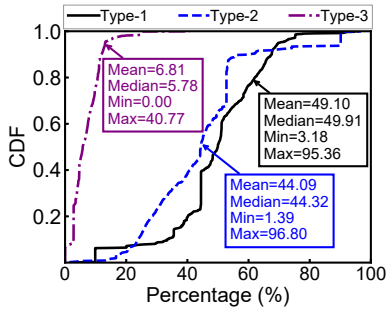


Figure 4: Percentages of specific types of API calls for the top-100 3D apps.

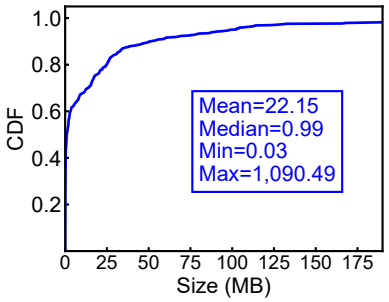


Figure 5: Graphics data amount generated per second by top-100 3D apps.

Graphics APIs beyond OpenGL. While the above descriptions focus on OpenGL (ES), we find that the API semantics of other existing graphics frameworks (such as Vulkan) have similar characteristics. Their APIs can also be categorized into the aforementioned three types. For example, in Vulkan `VKInstance` is used for managing context information, `vkCreateBuffer` is called for allocating buffer resources, and `vkCmdDraw` issues drawing commands.

This is not surprising, but stems from a common GPU's internal design. Like a CPU, a GPU usually leverages dedicated *state registers* for determining the current operation targets and parameters (*i.e.,* contexts), based on which an array of *computation cores* perform rendering and computing tasks in parallel. Special high-bandwidth *graphics memory* is often embedded in a GPU for holding a large amount of graphics resources (*e.g.,* vertex and texture), therefore mitigating the *memory wall* issue observed in a CPU [53], *i.e.,* the speed disparity between memory accesses and computations. Correspondingly, the three types of graphics API calls are then used for manipulating these essential hardware components throughout a rendering thread's lifecycle.

2.2 Real-World Graphics Workloads

To obtain a deeper understanding of modern graphics workloads in terms of both control flow and data flow, we measure the top-100 3D apps (which are all game apps) from Google Play as of 11/20/2021 [51] by examining the distributions of their API calls and the sizes of their generated graphics data. We instrument vanilla Android 11's system graphics library to log the API calls and count the graphics data of a test app during its run time. For each game app, we play a full game set (whose specific operations depend on the app's content) to record the runtime API invocation data. The experiments are conducted on a (middle-end) Google Pixel 5a device, which is equipped with a Qualcomm Snapdragon 765G SoC, 6 GB memory, 128 GB storage, and 1080p display.

Figure 3 shows that an average of 2,187 API calls are issued for rendering a single frame. For most (88%) of the frames, the number of API calls is larger than 1,000. Figure 4 depicts the percentages of specific types of API calls. As shown, the distribution is quite skewed—Type-1 and Type-2 occupy the vast majority (around 94% on average), while Type-3 take up merely 6% on average. Additionally, we find that despite being the majority, most Type-1 and Type-2 calls do not have immediate effects on the final rendering results until Type-3 calls are issued. For example, graphics data stored in a graphics buffer are usually not used by the GPU before certain drawing calls are issued.

With respect to data flow, there also exists considerable disparity in the graphics data amount generated per second, as indicated in Figure 5. While 90% of the graphics data generated per second are less than 60 MB in size, the peak data rate can be as high as 1.06 GB/second, revealing significant data rate dynamics in real-world graphics workloads.

2.3 Implications for Mobile Emulation

Type-1 and Type-2 calls are relatively cheap when executed natively, but this may not be the case in a virtualized environment. If a Type-1 or Type-2 call is synchronously executed on the host GPU, it can be expensive to first exit the guest, then wait for the host to execute the call, and then return back to the guest. This "tromboning" process adds substantial latency to what might otherwise be an inexpensive call, especially when Type-1 and Type-2 calls are very frequent.

To mitigate the problem, an intuitive approach is using a buffer to batch void API calls, *i.e.,* calls that do not return any values, so that not only the void Type-1 and Type-2 calls are delayed, but the asynchronous nature of Type-3 calls (which are all void calls) can also be exploited. However, the resulting efficiency improvement is limited by the proportion of void API calls, *i.e.,* only 41.4% according to our measurement. Thus, it is no wonder that GAE, which takes this approach to improve efficiency, cannot smoothly run many common apps.

In hopes of fundamentally addressing the problem, we make the following key observation—resource-related operations (involving all Type-2 and most Type-1 operations) are fully *handle-based*. That is to say, these operations only interact with indirect, lightweight resource handles in the main

memory, rather than the actual resources lying in the GPU's graphics memory. As demonstrated in Figure 2, a resource handle is merely an unsigned integer. In hardware GPU environments, this greatly facilitates the manipulation of graphics resources (without actually holding them in the main memory), thus avoiding frequently exchanging a large volume of graphics data between the main memory and the graphics memory. Note that the two memories are isolated hardware components connected via a relatively slow PCI bus.

We can exploit this key insight to accelerate mobile emulation, given that guest and host are also isolated by virtualization. We "project" a selective subset of contexts and resource handles, which are necessary for realizing actual rendering at the host GPU, onto the address spaces of guest processes; the resulting contexts after projection are termed *shadow contexts*. With the help of shadow contexts and resource handles, most (void and non-void) APIs can be asynchronously executed at the host. Moreover, certain Type-1 and Type-2 API calls (mostly used for querying context and resource information) can be directly accomplished within the projection space, *completely* eliminating their execution at the host.

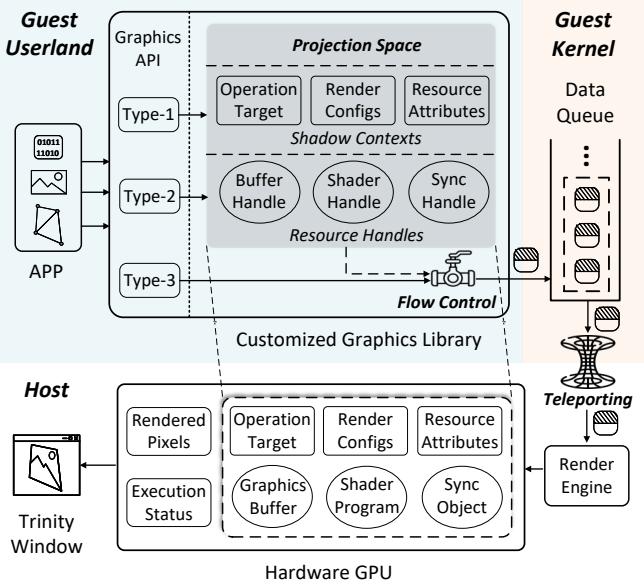


Figure 6: Architectural overview of Trinity.

3 System Overview

Figure 6 depicts Trinity's system architecture. It uses virtualization to isolate guest and host execution environments to retain strong compatibility and security. At the heart of Trinity lies a small-size *graphics projection space*, which is allocated inside the memory of a guest app/system process. Within the space, we maintain a special set of shadow contexts and resource handles which correspond to a subset of control contexts and resources inside a hardware GPU (*cf.* §4).

Once Type-1 or Type-2 API calls issued from a guest process are executed in the projection space, the shadow contexts and resource handles will reflect and preserve their effects. Control flow then returns to the guest process for executing its next program logic without synchronously waiting for host-side execution of the API calls (as conducted by API remoting). Meanwhile, the host contexts are asynchronously aligned with the shadow contexts; mappings are asynchronously established between resource handles and host resources.

Since synchronous host-side API execution is avoided, rather than exiting to the host to deliver data, the host can choose to asynchronously fetch the guest data required for API execution from the guest memory space through polling (*cf.* §6.1), thus reducing frequent VM Exits. Later when the guest process issues Type-3 API calls, they are also asynchronously executed at the host as they are designed to be asynchronous. In this manner, the originally time-consuming guest-host interactions can be effectively decomposed into interleaved and mostly asynchronous guest-projection interactions and projection-host interactions.

For example, when running the program in Figure 2, Trinity directly generates a buffer handle upon the Type-2 API call `glGenBuffers`, which is then sent to the host. When the program finishes sending the handle, its control flow continues; meanwhile, the host asynchronously allocates a buffer and its handle by also calling `glGenBuffers` in a dedicated host rendering thread using the host-side desktop OpenGL library, whose APIs are a superset of OpenGL ES APIs.

The relation between the host handle and the guest one is recorded in a hash table at the host. When `glBindBuffer` (Type-1) is called with the guest handle, Trinity adjusts the shadow context information of the currently bound buffer handle, and then sends the bound guest handle to the host. When the guest finishes sending the handle, the host asynchronously looks up the corresponding host handle in the hash table, and then calls `glBindBuffer` at the host to bind the host buffer (handle) in the rendering thread.

When `glMapBufferRange` (Type-2) is called, Trinity allocates a guest memory space and returns it to the guest program. When `glUnmapBuffer` (Type-2) is called, Trinity transfers the data in the guest memory space to the host, as no further modifications can be made to the data then. At the host side, the real buffer is then asynchronously populated with the data also through `glMapBufferRange`. Finally, upon `glDrawArrays` (Type-3), Trinity asynchronously executes it at the host rendering thread, so as to instruct the host GPU to realize actual rendering with the graphics buffer's data.

To sum up, Trinity's projection space provides two key advantages. First, it helps to avoid synchronous host-side execution of APIs (as in API remoting), even for non-void calls (such as `glGenBuffers`) that need to be processed immediately, so that expensive VM Exits can also be reduced. Second, it can resolve the API calls for querying context and resource information, such as `glGetBufferParameteriv`

in Figure 2, without sending them to the host. Quantitatively, 99.93% calls do not need synchronous host-side API execution, among which 26% are directly resolved at the guest (*cf.* §8.3). Although the projection space can involve processing certain calls twice—once at the guest and once at the host, this is done with relatively cheap operations whose extra costs are more than outweighed by the savings from reduced synchronous host-side execution of the APIs and the accompanied VM Exits.

To maximize Trinity's graphics processing throughput, all the above guest-side and host-side operations are coordinated by an elastic flow control algorithm (§5). Furthermore, the projection-host interactions are accomplished via a data teleporting method (§6) that attempts to maximize the data delivery throughput under high data and system dynamics.

4 Graphics Projection

We present the construction and maintenance of shadow contexts (§4.1) and resource handles (§4.2), *i.e.,* the key data structures that format the projection space.

4.1 Shadow Context

In §2.1, we have introduced that Type-1 APIs are usually used to manipulate three types of context information: 1) operation target, 2) render configurations, and 3) resource attributes. Apart from the above, as shown in Figure 6, context information in a real GPU environment also includes 4) rendered pixels and 5) execution status. Here rendered pixels refer to the rendered pixels stored in graphics memory, and execution status is the current status of the GPU's command queues.

For a shadow context, we carefully select to maintain the following three types of context information: 1) operation target, 2) render configurations, and 3) resource attributes. Consequently, with the above information, subsequent reads of context information can be directly fulfilled with the shadow contexts without resorting to the host. The shadow context is maintained based on Type-1 calls issued by a guest process. For example, when the process calls `glBindBuffer` (as shown in Figure 2) to bind a buffer handle (`vertex_buffer_handle`) as the current operation target, the operation target maintained in the shadow context (usually an integer) will be modified to the buffer handle.

The other two pieces of context information we choose not to maintain, *i.e.,* rendered pixels and execution status, are related to a hardware GPU's internal states. Managing such information requires frequent interactions with the host GPU, thus incurring prohibitively high overhead. If such information is actually required, it will be retrieved from the host synchronously. Fortunately, such cases occur with a pretty low (0.07% on average) probability during an app's rendering (according to our measurement in §2.2). Even when such cases occur, we make considerable efforts to minimize the incurred time overhead by carefully designing the data teleporting method, which will be detailed in §6.

Similar to a CPU context, a rendering context is tightly coupled with the thread model of an OS. At any given point of time, a thread is bound to a single rendering context, while a rendering context can be shared among multiple rendering threads of a process to realize cooperative rendering. Thus, in the graphics projection space of a process, we maintained shadow contexts on a per-thread basis, while keeping a reference to the possible shared contexts.

4.2 Resource Handle

As introduced in §2.1, resources involved in graphics rendering include *graphics buffers*, *shader programs* and *sync objects*. Compared to contexts, the allocation of resource handles and management of actual resources often require more judicious data structure and algorithm design, as well as guest-host cooperation, since they can easily induce inefficient memory usage and implicit synchronization, thus impairing system performance.

Handle Allocation. As mentioned before, all the graphics resources are managed through resource *handles* by modern GPUs. Guided by this, when a guest process requests for a resource allocation, we directly return a handle generated by us, which is not backed with a real host GPU resource upon handle generation. Then, after the control flow is returned to the guest process, the host will perform actual resource allocation in a transparent and asynchronous manner, and record the mapping between the guest handle and the host one in a host-side hash table. To make the guest-side handle allocation efficient, we adopt a bitmap for managing each type of resource handle, with which all the resource creation and deletion can be done in $O(1)$ time complexity, and we can maintain good memory density through handle recycling.

Resource Management. After allocating resource handles for a guest process, we also need to properly manage the actual resources underlying the allocated handles. In particular, the management of buffer resources is critical to system performance as they hold most of the graphics data. As discussed in §2.1, there are two approaches to populating a graphics buffer with data, *i.e.,* immediate copy and latent mapping.

For the former, developers would call `glBufferData` and pass the data's memory address to the API to initiate copying the data from the main memory to the graphics buffer. In this case, we need to immediately transfer the data (upon the API call) to the host as required by the API. For the latter, as discussed in §3, the data transfer is conducted when the guest memory space is unmapped (*i.e.,* `glUnmapBuffer` is called) by the guest process. When the data are transferred to the host, we need to populate the actual host-side graphics buffer with the data. To this end, we first ensure that the host context is aligned with the shadow context so that the correct

buffer is bound and populated. Then, to efficiently populate the buffer, we copy the data to a graphics memory pool we maintain at the host, which maps a pre-allocated graphics memory space to a host main memory address also using latent mapping. In this way, modern GPUs' DMA copy engine can still be fully utilized to conduct asynchronous graphics buffer population without incurring implicit synchronization (*cf.* §2.1). After this, the allocated guest memory space will be released, avoiding redundant memory usages.

5 Flow Control

With the guest and host control flows becoming mostly decoupled with the help of the projection space, their execution speeds also become highly uncoordinated. This is because a guest process' operations at the projection space usually only involve lightweight adjustments to the shadow contexts and resource handles, thus being much faster than host-side operations (*i.e.,* actual rendering using the hardware GPU).

At first glance, this should not raise any problems since guest API calls that require (synchronous or asynchronous) host-side executions can simply queue up at a guest blocking queue—if the queue is filled up, the guest process would block until the host render engine finishes prior operations. However, we find that in practice this could easily lead to *control flow oscillation*. From the guest process' perspective, a large amount of API calls are first quickly handled by the projection space when the data queue is not full. Soon, when the queue is filled up, a subsequent call would suddenly take a significantly longer time to complete as the queue is waiting for the (slower) host-side actual rendering. The long processing time further leads to a long delta time of the current frame as discussed in §2.1. As a result, the guest process may generate abnormal animations following the delta timing principle, *e.g.,* a game character could move an abnormally long distance in just one frame due to the long delta time, leading to poor user-perceived smoothness.

To resolve this problem, instead of solely relying on a blocking queue, we orchestrate the execution speeds of control flows at both the guest and host sides. Our objective is the *fast reconciliation* of the guest-side and host-side control flows, so that the overall performance of Trinity can be staying at a high level. To this end, we design an elastic flow control algorithm based on the classic MIMD (multiplicative-increase/multiplicative-decrease) algorithm [31] in the computer networking area, which promises fast reconciliation of two network flows. To adapt MIMD to our graphics rendering scenario, we regulate control flows' execution speeds at the fine granularity of each rendered frame.

In detail, when a guest rendering thread finishes all the graphics operations related to a frame's rendering, we let it sleep for T_s milliseconds and wait for the host GPU to finish the actual rendering. T_s is then calculated as $T_s = \frac{N'}{N} \times (\overline{T_h} - \overline{T_g})$, where N' is the current difference in the number of rendered frames between the guest's and host's rendering threads, N is the desirable maximum difference set by us (N is currently set to 3 in Trinity as we use the widely-adopted *triple buffering* mechanism for smooth rendering at the host), $\overline{T_h}$ is the host's average frame time (for executing all the graphics operations related to a frame) for the nearest N frames, and $\overline{T_g}$ is the guest's average frame time also for the nearest N frames. $\overline{T_h}$ and $\overline{T_g}$ are calculated by counting each frame's rendering time at the host and the guest sides.

Specifically, if $N' > N$ (*i.e.,* the guest is too fast), T_s will be multiplicatively increased to a longer time to approximate the host's rendering speed. Otherwise, T_s will be multiplicatively decreased, striving to maintain the current frame number difference at the desirable value. Typically, T_s lies between several milliseconds and tens of milliseconds depending on the guest-host rendering speed gap. In this way, Trinity can quickly reconcile the guest-side and host-side control flows.

6 Data Teleporting

Fast guest-host data delivery is critical for keeping projection-host interactions efficient. To realize this, we first analyze system and data dynamics (§6.1) that constitute a major obstacle to the goal, and then describe the workflow of our data teleporting method (§6.2), which leverages static timing analysis to accommodate the dynamic situations.

6.1 System and Data Dynamics

When control flows are synchronously accompanied by data flows, the guest-host data delivery mechanism can be very simple. For example, in API remoting, VM Exits/Enters are leveraged to achieve control handover and data exchange at the same time. In Trinity, however, data flows are decoupled from control flows (thanks to the graphics projection space), so we are confronted with complicated situations as well as design choices. Among these data flows, projection-host data exchanges are the most likely to become a performance bottleneck due to their crossing the virtualization boundary.

By carefully analyzing the projection-host data exchanges when running top-100 3D apps, we find that the major challenge of rapidly delivering them lies in the high dynamics of system status and data volume (abbreviated as *system dynamics* and *data dynamics* respectively). With regard to system dynamics, the major impact factors are the available memory bandwidth and current CPU utilizations, which are not hard to understand. As to data dynamics, call data of APIs that require synchronous host execution are sensitive to end-to-end latency (*i.e.,* the delay until host-side executions of the calls), while asynchronous ones require high processing throughput. Further, we pay special attention to distinct data sizes and bursty data exchanges (*i.e.,* bulk data exchange during a short period of time) which are common in modern graphics workloads as

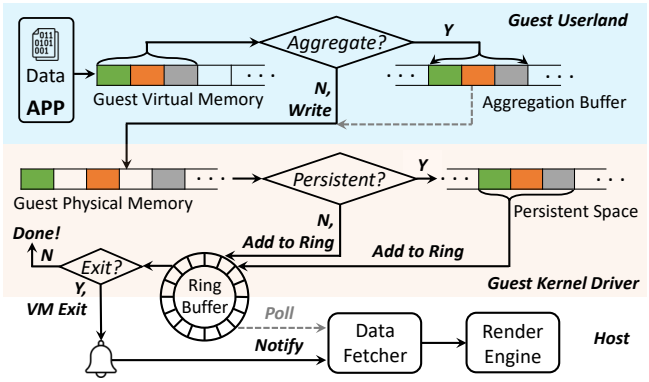


Figure 7: Workflow of data teleporting.

shown in Figure 5. In general, we can classify the dynamic situations into ∼16 patterns, roughly corresponding to the combinations of 1) high/low CPU utilization, 2) high/low available memory bandwidth, 3) synchronous/asynchronous API call data, and 4) large/small data sizes.

To accommodate the dynamic situations, our key observation is that the guest-host data delivery process can be decomposed into three stages, *i.e., data aggregation*, *data persistence* and *arrival notification*, as the data travel through the guest user space, the guest kernel space and the host. Moreover, in each of the stages, we find that there are mainly two different data delivery strategies, which make opposing tradeoffs under different dynamic situations as discussed below.

- Data Aggregation. As exercised in GAE, aggregating non-void API calls with a user-space buffer can usually reduce the frequency of user/kernel switches. This is also the case for Trinity since host-side execution of API calls is mostly asynchronous. However, if the data to be transferred are particularly large (*e.g.,* in bursty data exchanges), memory copies during data aggregation could bring larger time overhead compared to user/kernel switches; hence, the data should be delivered to the kernel as early as possible without any aggregation.
- Data Persistence. For the data of a guest rendering thread, we need to ensure their persistence until they are fetched by the host. To this end, a simple strategy is blocking the thread's control flow until the data delivery is done (as adopted by GAE). In Trinity, we realize that there is an alternative strategy by using a special persistent space (*e.g.,* in the guest kernel) to maintain the guest thread's data, so that there is no need to block the thread's control flow. Intuitively, this strategy is most suited to small data delivery, which does not incur long-time memory copies.
- Arrival Notification. To notify the host to fetch the data that have arrived, we can simply leverage the VM Exit-based strategy (adopted by GAE), whose incurred delays can be as low as tens of microseconds. This, however, can lead to the guest core's being completely stopped. Alternatively, for asynchronous data fetching, we can utilize a data polling-based strategy at the host, which does not incur the guest world's stopping but would introduce millisecond-level delay due to the thread sleeping and CPU scheduling delays of a common time-sharing host OS.

6.2 Workflow

Given that there is no single strategy that can accommodate every dynamic situation, we implement in Trinity all the combinations of strategies. Almost all of them are implemented at the guest side, except that data polling is realized by the host.

To decide the proper strategy during each stage of data delivery, we adopt the *static timing analysis* [12] method, which calculates the expected delay of each *timing step* (*i.e.,* stage) incurred by different data delivery strategies. As mentioned before, the stages include data aggregation, data persistence, and arrival notification. Suppose a guest app wishes to deliver a data chunk of size S_{data}, the current copy speed of the guest memory is V_{guest}, the current copy speed of the host memory is V_{host}. Below we elaborate on the workflow of data teleporting which selects the suitable strategy in each data delivery stage based on static timing analysis.

Data Aggregation. As shown in Figure 7, if the data to be delivered are asynchronous API call data (*i.e.,* call data of APIs that do not need synchronous host-side execution), we can aggregate them in a user-space buffer to reduce projection-host interactions. However, aggregating the data in the buffer incurs a memory copy, resulting in a delay of $\frac{S_{data}}{V_{guest}}$. Otherwise, an individual `write` system call will be invoked to write the data to our kernel character device driver (*cf.* §7), whose time overhead is T_{write}. Obviously, if $\frac{S_{data}}{V_{guest}} < T_{write}$, we choose to aggregate the data; else, we choose not to.

In contrast, for synchronous API call data we should always avoid data aggregation since synchronous calls should be immediately delivered to the host for executions. Then, along with these non-aggregation data, the aggregation buffer will also be written to our kernel driver and then cleared. We next enter the data persistence stage.

Data Persistence. In this stage, our kernel driver will decide whether to block the guest app's control flow, or utilize an additional persistent space for ensuring the persistence of a guest thread's data until the data are fetched by the host. Unlike the user-space data aggregation buffer that serves to reduce the frequency of entering the kernel and interacting with the host, the kernel persistent space allows the app's control flow to quickly return to the user space for executing its next logic. In practice, if we resort to the control flow blocking strategy, the blocking time will consist of four parts: 1) the delay of adding the data to a ring buffer shared by the guest and the host for realizing data delivery—T_{ring}, 2) the delay of host notification—T_{hn}, 3) the time for a host-side memory copy to fetch data (detailed later in Data Fetching)—$\frac{S_{data}}{V_{host}}$, and 4) the

delay of host-to-guest notification through interrupt injection for returning the control flow to the guest app—T_{gn}. Here the ring buffer does not directly store the data; instead, to transfer a large volume of data, it holds a number of (currently 1024) pointers, each of which points to another ring buffer of the same size, whose buffer item stores the data's physical addresses. Therefore, the blocking strategy's time overhead $T_{blocking}$ is the sum of them: $T_{blocking} = T_{ring} + T_{hn} + \frac{S_{data}}{V_{host}} + T_{gn}$ Here we encounter a challenge: T_{hn} is dependent on the arrival notification strategy which we have not decided yet. Fortunately, we find that when the control flow blocking strategy is adopted, the app thread's execution flow has already stopped. Thus, a VM Exit's side effect no longer matters in this case, but its advantage of short delay makes it an appropriate choice. We then naturally take the VM Exit-based arrival notification strategy, so T_{hn} generally equals the delay of a VM Exit.

On the other hand, if we choose to leverage a kernel persistent space for data persistence, the time overhead comes from 1) a memory copy to the persistent space and 2) adding the data to the ring buffer, *i.e.,* $T_{persistent} = \frac{S_{data}}{V_{guest}} + T_{ring}$. After the above are finished, the guest app's control flow is immediately returned to its user space for executing its next program logic, while the host asynchronously polls for data arrival and fetches data (as to be detailed later).

Based on the calculated $T_{blocking}$ and $T_{persistent}$, we can then choose the data persistence strategy with a smaller delay. Also, for synchronous API call data, we directly choose the blocking strategy because during synchronous calls the control flow is naturally blocked until host-side executions. With respect to the parameters used in the above analysis, they can be either directly obtained (*e.g.,* S_{data}) or statistically estimated by monitoring their recent values and calculating the average (*e.g.,* V_{guest} and V_{host}).

Arrival Notification. After the data are added to the ring buffer, we then need to choose a proper strategy for notifying the host of data arrival. In practice, we find that the arrival notification strategy is closely related to the data persistence strategy. Specifically, control flow blocking is particularly sensitive to the arrival notification delay, and thus should be coupled with VM Exits. On the contrary, the persistent space-based strategy allows arrival notification and data fetching to be asynchronous, and thus the polling-based strategy should be selected; the polling is performed by a host-side data fetching thread (referred to as Data Fetcher) every millisecond.

Data Fetching. When Data Fetcher is notified of data arrival, it would read the ring buffer to acquire the data. If the data are contiguous in the guest physical memory (and thus contiguous in the host virtual memory), the data can be directly accessed without further memory copy; otherwise, they should be copied to a contiguous host buffer. The fetched data are then distributed to the host render engine's rendering threads for realizing actual rendering.

7 Implementation

To realize Trinity, we make multiple modifications to the guest Android system and QEMU. First, we find that Android (as well as many UI-centric systems) clearly separates its versatile user-level graphics frameworks/libraries [6, 49] from the underlying system graphics library that realizes actual rendering. This enables us to effectively delegate every graphics API call by customizing only the system graphics library. At the guest user space, we replace the original system graphics library (*i.e.,* `libGLES`) with our customized one, which maintains the projection space and conducts flow control. The library exposes the standard OpenGL ES interfaces to apps, allowing them to seamlessly run without modifications.

To execute the delegated Type-1 and Type-2 APIs in the projection space, we implement all of them in the system graphics library, involving a total of 220 Type-1 APIs, 128 Type-2 APIs and 10 Type-3 APIs, which fully cover the standard OpenGL ES APIs from OpenGL ES 2.0 to the latest OpenGL ES 3.2. Additionally, we implement all the 54 Android Native Platform Graphics Interface (EGL) [5] functions to interface with the Android native window system. In practice, many APIs have similar functions, simplifying their implementations, *e.g.,* `glUniform` has 33 variants used for data arrays of different sizes and data types, such as `glUniform2f` for two floats and `glUniform3i` for three integers.

At the guest's kernel space and the host, we realize data teleporting via a QEMU virtual PCI device and a guest kernel driver. As a typical character device driver, our kernel driver mounts a device file in the guest filesystem, where the user-space processes can read from and write to so as to achieve generic data transferring. With this, API calls that require host-side executions are compacted in a data packet and distributed to our host-side render engine. The render engine then leverages the desktop OpenGL library to perform actual rendering using the host GPU.

Trinity is implemented on top of QEMU 5.0 in 118K lines of (C/C++) code (LoC). In total, the projection space, flow control and data teleporting involve 113K LoC, 220 LoC and 5K LoC, respectively. Among all the code, only around 2K LoC are OS-specific, involving kernel drivers and native window system interactions.

Trinity hosts the Android-x86 system (version 9.0). Since our modifications to QEMU and Android-x86 are dynamic libraries and additional virtual devices, they can be easily applied to higher-version QEMU and Android. Trinity can run on most of the mainstream OSes (*e.g.,* Windows 10/11 and macOS 10/11/12) with both Intel and AMD x86 CPUs. It utilizes hardware-assisted technologies (*e.g.,* Intel VT and AMD-V) for CPU/memory virtualization. For the compatibility with ARM-based apps, Trinity incorporates Intel Houdini [29] into the guest system for dynamic binary translation.

8 Evaluation

We evaluate Trinity with regard to our goal of simultaneously maintaining high efficiency and compatibility. First, we describe our experiment setup in §8.1. Next, we present the evaluation results in §8.2, including 1) Trinity's efficiency measurement with standard 3D graphics benchmarks, 2) Trinity's smoothness situation with the top-100 3D apps from Google Play, and 3) Trinity's compatibility with 10K apps randomly selected from Google Play. Finally, we present the performance breakdown in §8.3 by removing each of the three major system mechanisms—projection space, flow control and data teleporting.

8.1 Experiment Setup

To understand the performance of Trinity in a comprehensive manner, we compare it with six mainstream emulators, including GAE, QEMU-KVM, VMware Workstation, Bluestacks, and DAOW, as well as Windows Subsystem for Android (WSA)—a Hyper-V-based emulator released in Windows 11. Their architectures and graphics stacks are shown in Table 1. We use their latest versions as of Dec. 2021.

Software and Hardware Configurations. Regarding the configurations of these emulators, we set up all their instances with a 4-core CPU, 4 GB RAM, 64 GB storage, and 1080p display (*i.e.,* the display width and height are 1920 pixels and 1080 pixels, respectively) with 60 Hz refresh rate. However, since WSA does not allow customizing configurations, we use its default settings which utilize the host system's resources to the full extent. For other options (*e.g.,* network) in the emulators, we also leave them as default.

Our evaluation is conducted on a high-end PC and a middle-end PC. The former has a 6-core Intel i7-8750H CPU @2.2 GHz, 16 GB RAM (DDR4 2666 MHz), and a NVIDIA GTX 1070 MAX-Q dedicated GPU. The latter has a 4-core Intel i5-9300H CPU @2.4 GHz, 8GB RAM (DDR4 2666 MHz), and an Intel UHD Graphics 630 integrated GPU. Their storage devices are both 512 GB NVME SSD. Regarding the host OS, we run most of the abovementioned emulators on Windows 11 (latest stable version) given that WSA, Bluestacks, and DAOW are Windows-specific. However, since QEMU-KVM is Linux-specific, we run it on Ubuntu 20.04 LTS which is also the latest stable version as of Dec. 2021.

Workloads and Methodology. We use three different workloads to drive the experiments, in order to dig out the multi-aspect performance of Trinity. First, we use representative 3D graphics benchmark applications: 3DMark [34] and GFXBench [32], both of which are widely used for evaluating mobile devices' GPU performance. Together they provide three specific benchmarks, which are referred to as Slingshot Unlimited Test 1 (3DMark), Slingshot Unlimited Test 2 (3DMark) and Manhattan Offscreen 1080p (GFXBench).

Table 1: Comparison of the evaluated emulators.

Mobile Emulator	System Architecture	Graphics Stack
GAE [23]	x86 Android on customized QEMU	API remoting
WSA [41]	x86 Android on Windows Hyper-V	API remoting
QEMU-KVM [46]	Android-x86 on QEMU	Device emulation
VMware Workstation [52]	Android-x86 on VMware Workstation	Device emulation
Bluestacks [14]	Android-x86 on VirtualBox	Proprietary
DAOW [55]	Direct Android emulation on Windows	API translation with ANGLE [22]

These benchmarks generate complex 3D scenes in an *off-screen* manner, *i.e.,* the rendering results are not displayed on the screen and thus is not limited by the screen's refresh rate, so the graphics system's full potential can be tested. In detail, we run each benchmark on every emulator and hardware environment for five times, and then calculate the average results together with the error bars. Also, since the benchmarks come with Windows versions as well, we further run them directly on Windows to figure out the native hardware performance.

Second, to understand Trinity's performance on real apps, we run the top-100 3D (game) apps from Google Play as of 11/20/2021 [51], which are the same 100 apps discussed in §2.2. Concretely, for each of the apps, one of the authors manually runs a (same) full game set on every emulator, and repeats the experiment five times. During an app's running, we log the FPS (Frames Per Second) values of the app, which is a common indicator of a mobile system's running smoothness. We then use the average FPS value of the five experiments as the final FPS value of the app. Generally, we find that for all the studied apps, the standard deviations of the five experiments are all less than 4 FPS, indicating that the workloads are mostly consistent among different experiments. Since all the apps adopt the V-Sync mechanism to align their framerates with the screen's refresh rate (which is 60 Hz), their FPS values are always smaller than 60.

Third, to further evaluate Trinity's compatibility, we randomly select 10K apps from Google Play in Trinity. We use the *Monkey* UI exerciser [24] to generate random input events for each app for one minute, and monitor possible app crashes.

8.2 Evaluation Results

Graphics Benchmark. Figure 8 and Figure 9 illustrate the graphics benchmarks' results obtained on the high-end PC and the middle-end PC, respectively. Results of DAOW and WSA are not complete because they cannot successfully run all the benchmarks due to missing graphics APIs or abnormal API behaviors as complained by the benchmark apps. As shown, compared to the other emulators, Trinity can achieve the best efficiency on all the three benchmarks with both PCs.

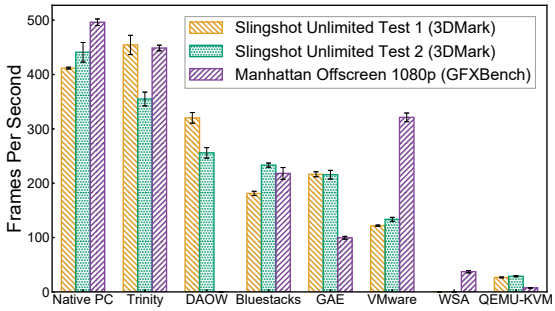


Figure 8: Benchmark results on the high-end PC.

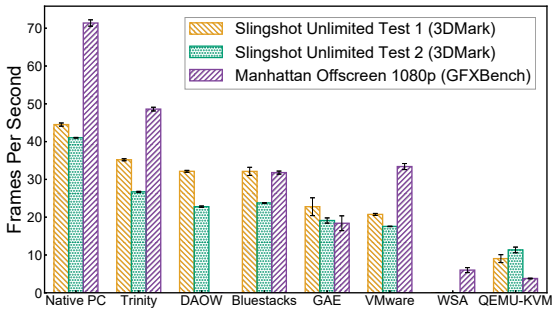
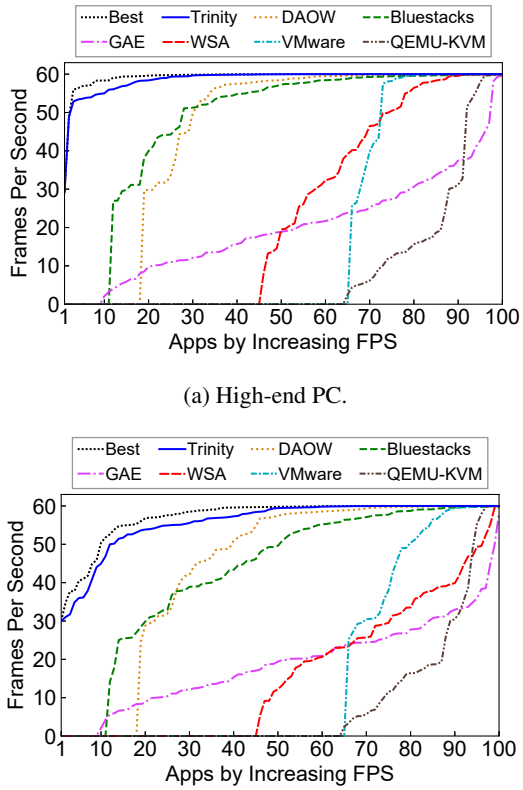


Figure 9: Benchmark results on the middle-end PC.



(a) High-end PC.

(b) Middle-end PC.

Figure 10: Average FPS of the top-100 3D apps across different emulators on the high-end and middle-end PCs. The "Best" line represents the highest FPS among the evaluated emulators of each app. If an app cannot run normally on an emulator, its corresponding FPS value is taken as zero.

Specifically, on the high-end PC that is equipped with a dedicated GPU, Trinity can outperform DAOW by an average of 40.5%, and reach 93.3% of the high-end PC's native hardware performance. In particular, for Slingshot Unlimited Test 1 we can achieve 110% native performance. This is attributed to the graphics memory pool (§4.2) maintained by Trinity at the host which can fully exploit the host GPU's DMA capability. Instead, the native version of the benchmark leverages synchronous data delivery into the GPU rather than a DMA-based approach, causing suboptimal performance. Further on the middle-end PC, we observe that Trinity can outperform the other emulators by at least 12.7%, indicating that Trinity can still maintain decent efficiency even on an integrated GPU with much poorer performance.

Top-100 3D Apps. Figure 10 depicts the average FPS of the top-100 3D apps from Google Play on different emulator platforms, when the apps are ranked by their FPS values on the corresponding emulator. Particularly, if an app cannot be successfully executed on (*i.e.,* is incompatible with) an emulator, its FPS value is taken as zero. Thus, the FPS values can reflect both the compatibility and efficiency of different emulators. In this regard, Trinity outperforms the other emulators by an average of 22.4%∼538% on the evaluated PCs. We next look into the compatibility and efficiency aspects of the evaluated emulators, respectively.

For compatibility, the numbers of compatible apps of Trinity, DAOW, Bluestacks, GAE, WSA, VMware, and QEMU-KVM are 100, 82, 89, 91, 55, 35 and 36, respectively. Delving deeper, we find that the root causes of other emulators' worse performance vary significantly. In detail, VMware and QEMU-KVM show the worst compatibility, mostly because their guest-side graphics stacks are both built atop the open-source desktop Linux graphics library Mesa [39], whose API behaviors sometimes differ from that of a typical Android graphics library. For GAE, its incompatibility with apps in fact roots in its poor efficiency—many incompatible apps become unresponsive for a long time during a game set, thus leading to Application Not Responding (ANR) [4]. For WSA, the problem is generally the same as GAE, as we find that WSA reuses most of the GAE's host-side and guest-side system components. Differently, its lack of Google Play Service (essential for many apps' running) in the guest system introduces more compatibility issues. For Bluestacks, its stable version runs an outdated Android 7.0 guest system, and thus cannot run some recent apps. Notably, despite the selective translation of system calls (*cf.* §1.1) that compromises compatibility, DAOW's compatibility with the 100 game apps is

only slightly worse than GAE, because it focuses on translating system calls frequently used by games [55].

For efficiency, we conduct a pairwise comparison between Trinity and each of the emulators in terms of the FPS of the apps that Trinity and the compared emulator can both successfully execute. On the high-end PC, Trinity outperforms DAOW, Bluestacks, GAE, WSA, VMware and QEMU-KVM in terms of the compatible apps by an average of 6.1%, 9.8%, 164.8%, 34.1%, 8.6%, and 132.2%, respectively. We observe a significant visual difference between Trinity and GAE, WSA, and QEMU-KVM across all apps. We observe less visual difference between Trinity and DAOW, Bluestacks, and VMware for many apps. However, the visual difference is very noticeable especially on apps where Trinity performs more than 15 FPS better, for which there were 9, 12, and 5 apps for DAOW, Bluestacks, and VMware, respectively. Regarding the average FPS values of individual apps, we find that Trinity shows the best efficiency on 76 of the apps. For the 24 apps that Trinity shows worse efficiency, we find that the differences in the apps' average FPS values are all less than 6 FPS, with 12 of them are in fact less than 1 FPS. On these apps, we find that there is not any notable smoothness difference between Trinity and the emulators that yield the best FPS.

Similar situations can also be observed on the middle-end PC (as demonstrated in Figure 10b). Trinity outperforms DAOW, Bluestacks, GAE, WSA, VMware and QEMU-KVM on the middle-end PC in terms of the compatible apps by an average of 4.9%, 16.1%, 168.7%, 84.6%, 17%, and 137.7%, respectively. Also, although there are more (42) apps where Trinity does not yield the best efficiency, the FPS differences are still mostly insignificant, with 36 of them being less than 5 FPS. For the remaining 6 apps, DAOW has the best FPS and outperforms Trinity by 6 to 9 FPS, though we could not perceive any visual difference between the two. Careful examination of the apps' runtime situations shows that they tend to heavily stress the CPU as its graphics scenes involve many physics effects such as collisions and reflections, which require the CPU to perform heavy computations such as matrix transformations. Thus, DAOW's directly interfacing with the hardware CPU without the virtualization layer allows it to perform better than Trinity (as well as the other emulators), particularly given the middle-end PC's rather weak CPU. In comparison, Trinity performs better than DAOW for all the 6 apps on the high-end PC.

Compatibility with Random 10K Apps. For the apps randomly selected from Google Play, we can successfully install all of them and run 97.2% of them without incurring app crashes. For the apps we cannot run, we find that some (2.3%) of them have also exhibited crashes on real devices; In addition, 0.43% require special hardware that Trinity currently has not implemented, *e.g.,* GPS, NFC and various sensors, which is not hard to fix given the general device extensibility of QEMU that Trinity is built on. Finally, the remaining 0.07% seem to actively avoid being run in an emulator by closing themselves when they notice that certain hardware configurations (*e.g.,* the CPU specification listed in `/proc/cpuinfo`) are that of an emulator as complained in their runtime logs.

8.3 Performance Breakdown

To quantitatively understand the contributions of the proposed mechanisms to Trinity's efficiency, we respectively remove each of the three major mechanisms of Trinity (*i.e.,* projection space, flow control and data teleporting), and measure the resulting efficiency degradations when running the top-100 3D apps on the high-end PC. In detail, removing projection space degrades Trinity to API remoting, whose guest-host control and data exchanges are still backed by our data teleporting mechanism. Removing data teleporting disables all the static timing analysis logics apart from data aggregation, which allows us to retain at least the data transferring performance of GAE since it also adopts a moderate buffer to batch void API calls. For data persistence and arrival notification, we adopt control flow blocking and VM Exit following GAE's design.

Further, to fully demonstrate the efficiency impacts of the three mechanisms, we also measure the performance breakdown when the maximum framerate restriction (which is 60 FPS) of the apps is removed. Note that we do not remove this restriction when evaluating the top-100 3D apps in §8.2 since this requires source code modifications to the emulators, while many of the emulators are proprietary (*e.g.,* DAOW and Bluestacks). Figure 11 depicts the average FPS values of the top-100 3D apps in the breakdown experiments with the 60-FPS framerate restriction, while Figure 12 shows the results without the framerate restriction.

Projection Space. After the projection space is removed, the average FPS drops by 6.1× (8.6×) with (without) the framerate restriction, providing the most significant efficiency benefits. This is not surprising as our in-depth analysis of the API call characteristics (by instrumenting our system graphics library as discussed in §2.2 during the breakdown experiments) shows that with the projection space, 99.93% of graphics API calls do not require synchronous host-side executions. The remaining 0.07% API calls are Type-1 calls related to the context information we do not maintain in shadow contexts, including the rendered pixels and execution status of a GPU as discussed in §4.1.

Among these asynchronously-executed calls, 26% are directly resolved at the projection space (with our maintained context and resource information), fundamentally avoiding their needs for any host-side executions. Such calls are mostly related to context manipulation and context/resource information querying. For the remainder (74%), they involve APIs for resource allocations and populations, as well as drawing calls. We also measure the memory consumption of the added projection space when running the top-100 3D apps by monitoring the maximum memory consumed by our provided

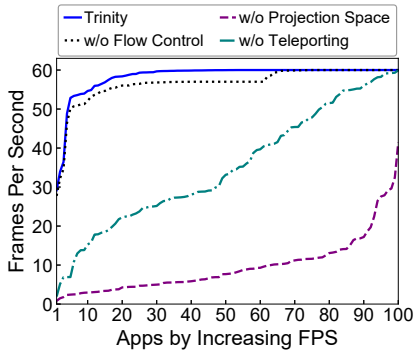


Figure 11: Performance breakdown with regard to the top-100 3D apps with framerate restriction.

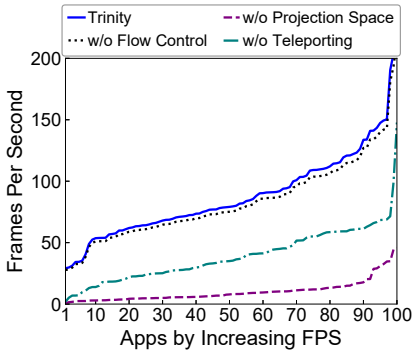


Figure 12: Performance breakdown with regard to the top-100 3D apps without framerate restriction.

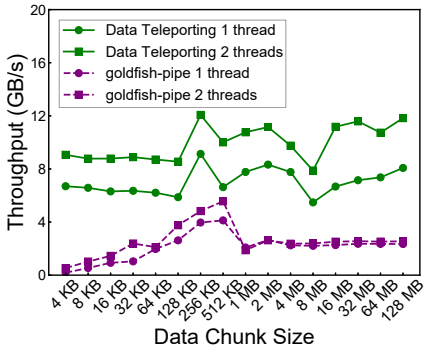


Figure 13: Throughput of data teleporting and `goldfish-pipe`, with one and two threads.

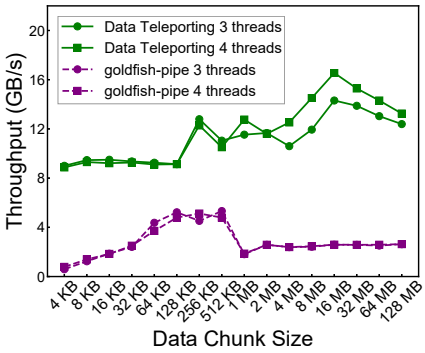


Figure 14: Throughput of data teleporting and `goldfish-pipe`, with three and four threads.

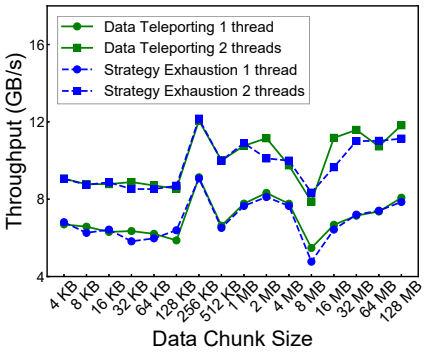


Figure 15: Throughput of data teleporting using strategy exhaustion and static timing analysis, with one and two threads.

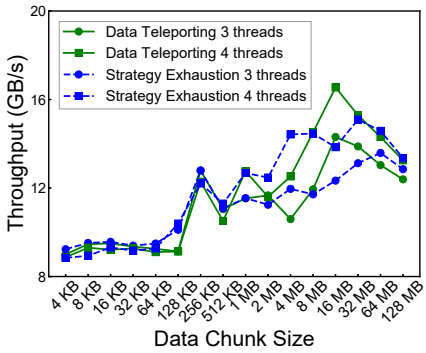


Figure 16: Throughput of data teleporting using strategy exhaustion and static timing analysis, with three and four threads.

system graphics library at the guest side. We find that the projection space only takes an average of 466 KB (at most 1021 KB) memory for an app. The memory consumption is small because the shadow contexts and resource handles are mostly small integers, and our careful resource management has prevented redundant memory usages.

Flow Control. On the other hand, flow control contributes 2.7% (5%) FPS improvement on average with (without) the framerate restriction. This is because flow control mainly serves to mitigate the control flow oscillation problem (*cf.* §5), thus contributing less to the running smoothness *as measured by FPS*. To quantify the actual effects of flow control, we further measure the occurrences of control flow oscillations during the apps' running. As a result, without flow control, control flow oscillation occurs 20× more frequently on average. When that happens, as discussed in §5, the apps' animations will look extremely unsmooth *from users' perspective* since many essential frames of the animations are skipped (*i.e.,* not rendered) by the apps as dictated by the delta timing principle, while the total number of frames rendered per second (*i.e.,* FPS) remains mostly unchanged.

Data Teleporting. Finally, when data teleporting is disabled, the fixed data delivery strategy cannot well adapt to system and data dynamics, leading to 1.7× (2.2×) FPS degradation with (without) the framerate restriction. To demystify the efficiency gains brought by data teleporting, we further examine its throughput under diverse system and data dynamics on the high-end PC. Specifically, we develop a benchmark app that synthesizes data chunks ranging from 4 KB (a continuous memory page space) to 128 MB, and doubles the size for each successive experiment. In each experiment, the app writes the data chunk to our kernel character device file (*cf.* §7) to transfer it to the host 1,000 times with one, two, three, or four threads; here the number of threads varies from one to four (the number of the emulator's CPU cores) to mimic different system dynamics. By measuring the time consumed for data transfer, we can calculate the final throughput result.

In comparison, we conduct the same experiments on GAE's guest-host I/O pipe called `goldfish-pipe`, which is GAE's core infrastructure for sending API call data from the guest to the host and realizing API remoting. To this end, we customize GAE to include a dedicated graphics API for throughput

measurement, which our benchmark app can call to transfer guest data to the host as described above. This API is made to be a void API so that GAE's buffer for batching void APIs can take effect. Consequently, as shown in Figure 13 and Figure 14, data teleporting's throughput clearly exceeds that of `goldfish-pipe` under all the data and thread settings. On average, data teleporting's throughput is 5.3 times larger than that of `goldfish-pipe`.

Furthermore, we wish to know the effectiveness of static timing analysis. For this purpose, we measure the performance of the data teleporting mechanism using the above experiments when we adopt every possible strategy. Then, we compare the highest throughput produced by the above strategy exhaustion with that produced by the static timing analysis. As shown in Figure 15 and Figure 16, the throughput values produced by strategy exhaustion and static timing analysis are very close (4% average deviation). More in detail, static timing analysis can make the most suitable strategy choice in 95.4% of the data delivery tasks.

9 Related Work

Commercial Mobile Emulators. A plethora of commercial mobile emulators have similar architectures to the ones we evaluate in §8. For instance, Anbox [3], which directly runs Android's Framework layer on a Linux PC, leverages the container technique to achieve lightweight guest-host isolation, and reuses GAE's graphics stack—all the guest-side graphics operations are sent to a host-side daemon for execution, thus requiring synchronous inter-process communications. Accordingly, its efficiency is similar to that of GAE.

LDPlayer [35], MEmu [38], NoxPlayer [42] and Genymotion [20] all adopt the AOVB (Android-x86 on VirtualBox) architecture (as in Bluestacks). To realize graphics rendering, they also reuse some of the graphics libraries of GAE, *e.g.,* `libGLESv2_enc` at the guest that encodes OpenGL ES API calls into a data packet, and ANGLE [22] at the host that translates guest-side OpenGL ES calls to desktop OpenGL or Direct3D calls. Prior measurements [13, 55] show that the performance of such AOVB-based emulators is close to that of Bluestacks, probably due to their similar architectures.

GPU Virtualization. In PC/server virtualization, GPU multiplexing is typically achieved through hardware-assisted GPU passthrough [1, 2] or mediated passthrough [27, 30, 43], which allow a virtual machine (VM) to directly access the host GPU by remapping its DMA channels and interrupts to the guest. Differently, GPU passthrough monopolizes the host GPU, while the mediated approach allows sharing the GPU among multiple VMs through GPU context isolation.

However, the substantial differences between the graphics stacks of desktop OSes and mobile OSes significantly hinder their adoption by mobile emulators, as host GPUs' drivers are missing in mobile systems and developing them for mobile environments is extremely complicated (since mainstream desktop GPUs' specifications are often proprietary). Hence, we take a completely different approach of graphics projection to address the problem of multiplexing the host GPU, which is agnostic to the underlying hardware specifications and thus should also be beneficial to PC/server GPU virtualization.

Cross-OS and Cross-Device Graphics Stacks. Trinity focuses on Android emulation on a PC, while several researches have explored running iOS apps on Android graphics stacks based on their similarities in OpenGL ES libraries [7, 8]. This suggests that Trinity's graphics projection mechanism might also be applicable to the emulation of iOS apps on a PC. Also, various approaches remote graphics processing from one device to another over a network [9–11, 25, 47, 48]. For them, data exchanges over network often constitute a major bottleneck, which is similar to the bottleneck of frequent cross-boundary control/data exchanges in the virtualization setting. Thus, our idea of decoupling guest/host control and data flows via graphics projection should also be useful to relevant studies and applications, *e.g.,* cloud/edge gaming.

10 Conclusion

In this paper we present the design, implementation, performance, and preliminary deployment of the Trinity mobile emulator. It substantially boosts the efficiency of mobile emulation while retaining high compatibility and security through graphics projection, a novel approach that minimizes the coupling between the guest-side and host-side graphics processing. This unique design, together with strategic flow control and data teleporting, make Trinity a first-of-its-kind emulator that can smoothly run heavy 3D mobile games (achieving near-native hardware performance) and meanwhile retain comprehensive app support and solid guest-host isolation.

As part of a major commercial Android IDE, Trinity is expected to be used by millions of Android developers in the near future, contributing vibrantly to the ecosystem. We believe that many lessons and experiences gained from this work could also be applied to (graphics-heavy) PC emulation and cloud/edge gaming, as to be explored in our future work.

Acknowledgements

We would like to express our deepest appreciation to our shepherd, Jason Nieh, who was very responsive during our interactions with him and provided us with valuable suggestions, which have significantly improved our paper. We also thank the anonymous reviewers for their constructive suggestions. We thank Wei Liu and Xinlei Yang for their help in data collection and analysis. This work is supported in part by the National Key R&D Program of China under grant 2021YFB2900100, as well as the National Natural Science Foundation of China (NSFC) under grant 61902211.

A Artifact Appendix

Abstract

Trinity's artifact is publicly available at GitHub. To facilitate developing and using Trinity, we provide step-by-step instructions in the form of both documentations and videos. Please refer to our README file at https://github.com/TrinityEmulator/TrinityEmulator for details.

Scope

The artifact can be used to reproduce all the major results, including those of the graphics benchmarks and 3D apps.

Contents

Trinity's artifact includes code of the host emulator, binary of the guest Android system, and our evaluation scripts/data.

Hosting

We host the code/binary and data in two repositories (both in the `main` branch). We also provide a DOI for the artifact.

- **Trinity Code and Binary.**

  Link: https://github.com/TrinityEmulator/TrinityEmulator.

- **Evaluation Data and Figure Plotting Script.**

  Link: https://github.com/TrinityEmulator/EvaluationScript.

- **DOI for the Artifact.**

  DOI: 10.5281/zenodo.6586575

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*, 2006.

[2] AMD. I/O Virtualization Technology Specification Revision 1.26. *AMD White Paper*, 1:2–11, 2009.

[3] Anbox.com. Anbox: Container-based Android Emulator, 2021. https://anbox.io/.

[4] Android.org. Application Not Responding of Android, 2021. https://developer.android.com/topic/performance/vitals/anr.

[5] Android.org. GraphicBuffer: Android's Native Window Buffer Implementation, 2021. https://android.googlesource.com/platform/frameworks/native/+/jb-mr0-release/libs/ui/GraphicBuffer.cpp.

[6] Android.org. View: Basic Building Blocks for Android User Interface, 2021. https://developer.android.com/reference/android/view/View.

[7] J. Andrus, N. AlDuaij, and J. Nieh. Binary Compatible Graphics Support in Android for Running iOS Apps. In *Proc. of ACM/IFIP/USENIX Middleware*, pages 55–67, 2017.

[8] J. Andrus, A. Van't Hof, N. AlDuaij, C. Dall, N. Viennot, and J. Nieh. Cider: Native Execution of IOS Apps on Android. In *Proc. of ACM ASPLOS*, pages 367–382, 2014.

[9] Apple.com. AirPlay: Share Mutimedia Contents across Devices, 2021. https://www.apple.com/airplay/.

[10] R. A. Baratto, L. N. Kim, and J. Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proc. of ACM SOSP*, pages 277–290, 2005.

[11] R. A. Baratto, S. Potter, G. Su, and J. Nieh. MobiDesk: Mobile Virtual Desktop Computing. In *Proc. of ACM MobiCom*, pages 1–15, 2004.

[12] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer. Statistical Timing Analysis: From Basic Principles to State of The Art. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):589–607, 2008.

[13] Bluestacks.com. Benchmark Performance Comparisions among Bluestacks, LDPlayer, Memu, and Nox, 2021. https://www.bluestacks.com/bluestacks-vs-ldplayer-vs-memu-vs-nox.html.

[14] Bluestacks.com. Bluestacks: Modern Android Gaming Emulator, 2021. https://www.bluestacks.com/.

[15] T. Capin, K. Pulli, and T. Akenine-Moller. The State of the Art in Mobile Graphics Research. *IEEE Computer Graphics and Applications*, 28(4):74–84, 2008.

[16] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, 2012.

[17] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[18] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí. rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters. In *Proc. of IEEE HPC*, pages 224–231, 2010.

[19] A. Edmundson, R. Ensafi, N. Feamster, and J. Rexford. A First Look into Transnational Routing Detours. In *Proc. of ACM SIGCOMM*, pages 567–568, 2016.

[20] Genymotion.com. Genymotion: Android as a Service, 2021. https://www.genymotion.com/.

[21] L. Gong, Z. Li, F. Qian, Z. Zhang, Q. A. Chen, Z. Qian, H. Lin, and Y. Liu. Experiences of Landing Machine Learning onto Market-Scale Mobile Malware Detection. In *Proc. of ACM EuroSys*, pages 1–14, 2020.

[22] Google.com. Almost Native Graphics Layer Engine, 2021. https://github.com/google/angle.

[23] Google.com. Android Emulator: Simulates Android Devices on Your Computer, 2021. https://developer.android.com/studio/run/emulator.

[24] Google.com. Monkey: Automatic UI/Application Exerciser, 2021. https://developer.android.com/studio/test/monkey.

[25] Google.com. Stream Content with Chromecast, 2021. https://store.google.com/us/product/chromecast?hl=en-US.

[26] Google.com. SwiftShader: A CPU-Based Implementation of Graphics APIs, 2021. https://github.com/google/swiftshader.

[27] A. Herrera. NVIDIA GRID: Graphics Accelerated VDI with the Visual Performance of a Workstation. *NVIDIA Corp*, pages 1–18, 2014.

[28] Huawei.com. Huawei's DevEco Studio, 2021. https://developer.harmonyos.com/en/develop/deveco-studio/.

[29] Intel.com. Houdini: Translate The ARM Binary Code into the x86 Instruction Set, 2021. https://www.intel.com/content/www/us/en/products/docs/workstations/resources/accelerate-game-development-houdini-optane-memory.html.

[30] Intel.com. Intel GVT-g: Full GPU Virtualization with Mediated Pass-through, 2021. https://github.com/intel/gvt-linux/wiki/GVTg_Setup_Guide.

[31] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. In *Proc. of ACM SIGCOMM*, pages 83–91, 2003.

[32] Kishonti Ltd. GFXBench: A Unified Graphics Benchmark Based on DXBenchmark, 2021. https://gfxbench.com/.

[33] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proc. of the Linux Symposium*, volume 1, pages 225–230, 2007.

[34] U. Laboratories. 3DMark: Popular Benchmarks for Gamers, Overclockers, and System Builders, 2021. https://www.3dmark.com/.

[35] LDPlayer.com. LDPlayer: Free Android Emulator for PC, 2021. https://www.ldplayer.net/.

[36] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming. In *Proc. of ACM MobiSys*, pages 151–165, 2015.

[37] M. Li, H. Lin, C. Liu, Z. Li, F. Qian, Y. Liu, N. Sun, and T. Xu. Experience: Aging or Glitching? Why Does Android Stop Responding and What Can We Do About It? In *Proc. of ACM MobiCom*, pages 1–11, 2020.

[38] MEmu.com. MEmu: The Most Powerful Android Emulator, 2021. https://www.memuplay.com/.

[39] Mesa.org. The Mesa 3D Graphics Library, 2021. https://www.mesa3d.org/.

[40] Microsoft.com. Introduction to Hyper-V on Windows, 2021. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/.

[41] Microsoft.com. Windows Subsystem for Android, 2021. https://docs.microsoft.com/en-us/windows/android/wsa/.

[42] NoxPlayer.com. NoxPlayer: The Perfect Android Emulator to Play Mobile Games on PC, 2021. https://www.bignox.com/.

[43] NVIDIA.com. vGPU: Security Benefits of Virtualization as well as the Performance of NVIDIA GPUs, 2021. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[44] J. Oberheide and C. Miller. Dissecting The Android Bouncer. *SummerCon2012, New York*, 95:110, 2012.

[45] Oracle.com. VirtualBox: A Powerful x86 and AMD64/Intel64 Virtualization Product, 2021. https://www.virtualbox.org/.

[46] QEMU.org. QEMU: A Generic and Open Source Machine Emulator and Virtualizer, 2021. https://www.qemu.org/.

[47] RealVNC.com. VNC; Remote Desktop Access, 2021. https://www.realvnc.com/en/.

[48] S. Shi and C.-H. Hsu. A Survey of Interactive Remote Rendering Systems. *ACM Computing Surveys*, 47(4):1–29, 2015.

[49] Skia.org. Skia: 2D Graphics Rendering Library, 2021. https://skia.org/.

[50] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor?

[51] Trinity.github. List of Top-100 3D Apps, 2021. https://github.com/TrinityEmulator/EvaluationScript/#4-top-100-3d-apps.

[52] VMware.com. VMware Workstation Pros: Run Windows, Linux and BSD Virtual Machines on a Windows or Linux Desktop, 2021. https://www.vmware.com/products/workstation-pro.html.

[53] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[54] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *Proc. of ACM MobiSys*, pages 168–179, 2019.

[55] Q. Yang, Z. Li, Y. Liu, H. Long, Y. Huang, J. He, T. Xu, and E. Zhai. Mobile Gaming on Personal Computers with Direct Android Emulation. In *Proc. of ACM MobiCom*, pages 1–15, 2019.

ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs

Ashraf Mahgoub
Purdue University

Edgardo Barsallo Yi
Purdue University

Karthick Shankar
Carnegie Mellon University

Sameh Elnikety
Microsoft Research

Somali Chaterji
Purdue University

Saurabh Bagchi
Purdue University

Abstract

Serverless applications represented as DAGs have been growing in popularity. For many of these applications, it would be useful to estimate the end-to-end (E2E) latency and to allocate resources to individual functions so as to meet probabilistic guarantees for the E2E latency. This goal has not been met till now due to three fundamental challenges. The first is the high variability and correlation in the execution time of individual functions, the second is the skew in execution times of the parallel invocations, and the third is the incidence of cold starts. In this paper, we introduce ORION to achieve this goal. We first analyze traces from a production FaaS infrastructure to identify three characteristics of serverless DAGs. We use these to motivate and design three features. The first is a performance model that accounts for runtime variabilities and dependencies among functions in a DAG. The second is a method for co-locating multiple parallel invocations within a single VM thus mitigating content-based skew among these invocations. The third is a method for pre-warming VMs for subsequent functions in a DAG with the right look-ahead time. We integrate these three innovations and evaluate ORION on AWS Lambda with three serverless DAG applications. Our evaluation shows that compared to three competing approaches, ORION achieves up to 90% lower P95 latency without increasing $ cost, or up to 53% lower $ cost without increasing P95 latency.

1 Introduction

Serverless computing (*a.k.a.*, FaaS) has emerged as an attractive model for running cloud software for both providers and tenants. Recently, serverless environments are becoming increasingly popular for video processing [12, 58], machine learning [18, 55], and linear algebra applications [32, 48]. The requirements of these applications can vary from latency-strict (*e.g.*, Video Analytics for Amber Alert responders [61]) to latency-tolerant but cost-sensitive (*e.g.*, Training ML models [28]). Accurate latency estimation is essential to meet the requirements for both, as the cost in FaaS platforms is based on resource usage and runtime. The workflow of these serverless pipelines is usually represented as a directed acyclic graph (DAG) in which nodes represent serverless functions and edges represent data flow dependencies between them.

Serverless platforms experience high performance variability [4, 27, 35, 40, 42, 56] due to three primary reasons: First, some function invocations have cold starts. Second, there is skew in the execution time of various functions because of different content characteristics that the functions operate on. Third, there exists skew in the execution time due to variability in infrastructure resources (*e.g.*, network bandwidth for an allocated VM). Because of this variance in performance, predicting the mean (or median) execution time of individual functions is not sufficient to meet percentile-specific latency requirements (*e.g.*, P95) for serverless DAGs. Rather, a distribution-aware modeling technique is essential to capture this variability and provide accurate latency SLOs.

Key Idea. We propose ORION, a novel technique for performance modeling of serverless DAGs to estimate the end-to-end (E2E) execution time (synonymously, E2E latency). We leverage this model to enable system optimizations such as allocating resources to each function to reduce E2E latency while keeping $ costs low and utilization high. The different components of ORION are shown in Figure 1. We derive insights about serverless DAGs from analysis of production traces at *Azure Durable Functions* [13]. This analysis drives our performance model and the design features.

First, we observe the inherent performance variability in serverless DAGs and therefore represent the latency of a single function, as well as that of the entire DAG, as a distribution rather than a single value. For example, Figure 5 shows the variance in execution times for the top-5 most frequently invoked DAG-based applications. The execution times of invocations of the *same* DAG vary significantly, and the P95 latency is 80X of the P25 latency, averaged over the 5 applications. Thus, our performance model profiles the latency distribution for each function in the DAG and builds a performance model to capture the impact of varying the resource allocation to that function on its latency distribution. Afterward, we estimate the DAG E2E latency distribution by applying a se-

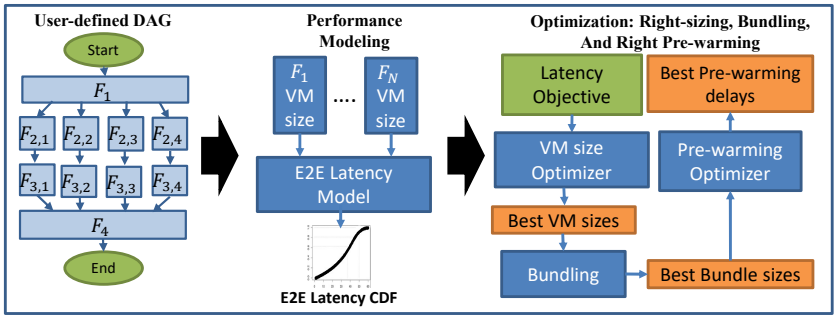


Figure 1: ORION *Overview.* ORION *profiles the DAG, estimates E2E latency CDF using CONV and MAX, and performs three system optimizations — right-sizing, bundling, and right pre-warming.*

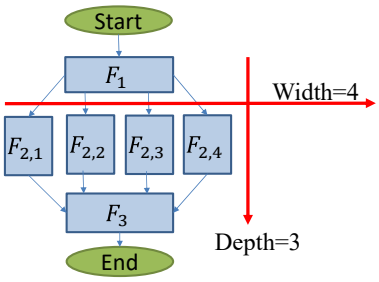


Figure 2: Illustration of DAG Depth **(i.e.,** *number of in-series stages) and Width* **(i.e.,** *Maximum fanout degree).*

ries of two statistical combination operations, `convolution` and `max`, for in-series and in-parallel functions, respectively. Moreover, we observe it is essential to consider the correlation between the workers across stages and within the same stage to accurately estimate the joint distributions. Our performance model does not require expensive profiling, as is needed for the leading technique, Bayesian Optimization [3, 5, 15].

We then use our performance model to design three optimizations for serverless DAGs. (1) **Right-sizing**: Finding the best resource configurations for each function to meet an E2E latency objective (*e.g.*, 95-th percentile latency < X sec) with the minimum cost. (2) **Bundling**: Identifying stages where co-locating multiple parallel instances of a function together to be executed on a single VM will be beneficial. The benefit arises when there is computation skew among the parallel workers caused by different content inputs. (3) **Right pre-warming**: The VMs to execute the functions in the DAG are pre-warmed just right, ahead of time, so that cold starts can be avoided while keeping provider-side utilization of resources high. With these three optimizations, ORION accurately meets latency SLOs while reducing execution cost (Figure 3).

ORION can be deployed by either the cloud service provider or by the end consumer. For the former, the use case is to provision its resources better to meet client SLOs. For the latter, the driving force is the appropriate resource provisioning to minimize E2E latency while minimizing execution cost.

Evaluation and Insights. We evaluate ORION on three serverless applications on AWS Lambda: two variants of Video Analytics [35], an ML Pipeline [17], and an NLP Chatbot [44] application. Our evaluation, comparing to three approaches: Best-Memory [2, 59], Speculative-Execution [30], and CherryPick [5], shows that the benefits of ORION persist across the different applications with different DAG structures, skews in execution times, and invocation frequencies. Our evaluation brings out the following insights:

(1) Latency correlation across stages *and* within a stage is important (Tables 1 and 2). Even when correlations are weak, not taking them into account can introduce significant error in the latency estimations. Further, to make the solution scalable, we have to compute the E2E latency estimation using just the right degree of correlation.

(2) Selecting the best VM sizes for serverless functions in a DAG is challenging (Table 3). This is because different resources in a VM scale up differently with their size. For example, for AWS-λ, CPU cores go from fractional to a maximum of six, network bandwidth only increases till a level, and disk capacity stays constant.

(3) Bundling multiple parallel instances of a function within a single VM helps when there is content skew and the functions are scalable, *i.e.*, they can make use of additional resources (Figure 16). Here also, the degree of bundling has to be carefully determined so as not to cause resource contention.

(4) Using the DAG structure and the function latency model, we can estimate the right time to pre-warm VMs and thus mitigate cold starts (Figure 15). With pre-warming, lower P95 latency is achieved with higher resource utilization.

In summary, the main contributions of ORION are the following: (1) Workload characterization for serverless DAGs seen by *Azure Durable Functions*. (2) A performance model for E2E latency of serverless DAGs; (3) A method for assigning the right resources for serverless DAGs to meet the E2E latency requirements within a reduced $ cost; (4) An application-independent way to bundle multiple function invocations to mitigate skews. (5) A method for deciding when to start pre-warming VMs for functions in a serverless DAG to minimize initialization latency while providing acceptable resource utilization.

ORION is open sourced and we release its code, the workload characterization data, and the evaluation applications at: https://github.com/icanforce/Orion-OSDI22

2 Motivation

2.1 Workload Characterization

Definitions. A DAG is a chain of two or more serverless function stages that execute in-series. A stage consists of one or more parallel invocations (a.k.a. instances) of the same serverless function. DAG *depth* is the number of stages in the DAG. DAG *width* is the max number of parallel worker functions (a.k.a. fanout degree) across all stages in the DAG.

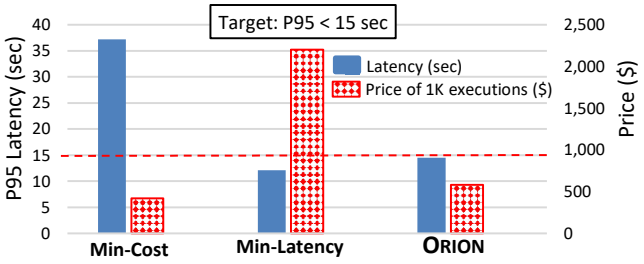


Figure 3: ORION ***improves both latency and cost of Video Analytics DAG. Executions with min VM size (i.e., Min-Cost) and max VM size (i.e., Min-Latency) are for reference, and all latencies are for warm executions.***

A DAG with $width = 1$ means it is a chain of sequential function invocations, whereas a DAG with $width > 1$ means it has at least one parallel stage. We show an illustration of DAG depth and width in Figure 2. Finally, we define skew in a parallel stage as the ratio of the execution times of the slowest to the fastest worker function.

Analysis. In this section, we characterize the workload of serverless DAGs from *Azure Durable Functions*. We collect a subset of the logs of DAG executions from six datacenters — three located in the US, two in Europe, and one in Asia from 10/19/21 to 10/25/21 (1 week). The workload we analyze includes 20M-30M DAG executions/day. From our characterization, we draw the following conclusions, which in turn motivate various design decisions of ORION.

(1) DAG Structure and Execution Time. We study the depth and width of serverless DAGs and their distributions in Figure 4. We account for the DAG invocation frequencies — if a DAG is invoked N times, its width and depth are included N times in the CDFs. First, we notice that DAG depth is low, with a median of 3 stages and a P95 of 8 stages. Second, although 65% of the DAGs are linear chains (no fanout), the width can grow to as high as 37 in the 95th percentile. We also study the execution time of DAGs and find that they can range from 10 ms to 112 min, with a median of 3.7 sec and a mean (weighted by invocation frequencies) of 48 sec. This motivates the need for considering DAG structure while minimizing the E2E latency.

(2) DAG Invocation Frequency and Relation to Cold Starts. Figure 7 shows the frequency of invocations/day for each DAG and the corresponding percentages of cold starts. We notice that the frequency of invocations is heavily skewed, with the top-5 most frequent DAGs accounting for 46% of all invocations. Thus, the optimized executions of these frequently invoked DAGs result in higher cost savings. We also notice that the percentage of cold starts decreases with higher invocation frequency. For example, DAGs invoked ≥ 100 times/day have very low percentages of cold start with a median of 0.35%. However, most of the DAGs are rarely invoked: 80% of the DAGs have an invocation frequency of < 100 times/day, and these experience a high percentage of cold starts with a median of 50%. This shows an increase in the proportion of infrequent serverless applications (DAG-based in our case) compared to a prior study [47], which showed that 55% of the serverless applications are invoked less than 100 times/day.

Hence, using keep-alive policies (as done by most major FaaS platforms) for those DAGs will not be sufficient and pre-warming becomes essential to mitigate cold starts. Even for the DAGs that are not the most frequently invoked, keeping E2E latency low is a desirable feature.

(3) Variance in DAG Execution Time. Figure 5 shows a boxplot for the execution time of the top-5 most frequently invoked DAGs (which contribute 46% of all invocations). We notice that the variance in execution times across different invocations of the same DAG is high. For example, the P95 latency is 80X the P25 latency, averaged over the five DAGs. We also notice that the *distribution of execution times can be heavily skewed.* For example, P50 is not centered between MIN and MAX, or between P25 and P75. Hence, it is essential to represent E2E latency of serverless DAGs as a distribution when modeling their performance to capture this variability.

(4) Degree of Skew. Figure 6 shows the skew distribution for parallel stages for different ranges of DAG widths. We notice that skew among parallel worker functions is at least 2× for 98.2% of the DAGs and increases as the width increases. This motivates the need for a mechanism to mitigate latency skew of parallel worker functions to reduce the E2E latency.

2.2 Performance Modeling

Modeling Latency as a Distribution rather than a Single Statistic. To estimate the E2E latency and cost of a serverless DAG, it is essential to model the latency of each component as a distribution. For example, the latency of the image classification function (`Classify-Frame`) in the Video Analytics DAG can vary by up to 2× when processing different frames, even when keeping the VM-size fixed to 1 GB. Although similar performance variability can be observed in server-centric platforms, our model is geared to serverless platforms due to their ability to scale resources according to demand virtually unboundedly and hence showing negligible queuing times [56]. Now consider a simple stage of two `Classify-Frame` functions running in parallel. Let X and Y be random variables representing the latency of each. The E2E latency of the two workers combined is given as $P(Z \leq z) = P(X \leq z, Y \leq z)$, which depends on the slowest of the two and hence knowing their median or even their P99 latencies is not sufficient to estimate their combined CDF. In fact, we need the *entire* distribution of *both* components to estimate the E2E latency distribution. Moreover, simply using statistical tail bounds is not suitable for our purpose. For example, Chebyshev's inequality uses the mean and variance to establish a loose tail bound and it is not known how to combine tail bounds for in-series and in-parallel functions with their correlations.

Modeling Correlation. To accurately estimate the combined latency distribution for in-series or in-parallel functions, we need to capture the correlation between their execution times.

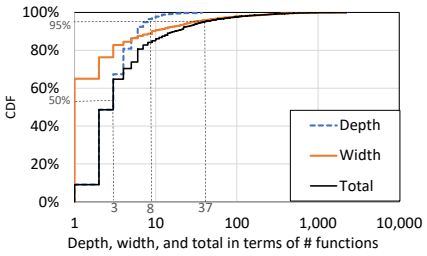


Figure 4: Characterization of depth (number of stages), width (degree of parallelism), and total number of nodes. Depth is low (P50 = 3; P95 = 8). 65% of DAGs are linear chains (no fanout) and width reaches 37 at the 95th percentile.

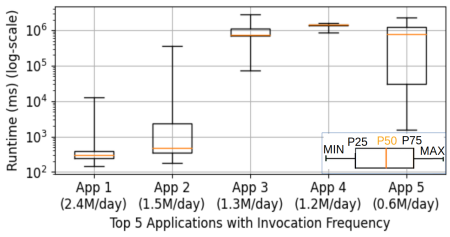


Figure 5: Latency distributions for the top-5 most frequent applications executed by **Azure Durable Functions** *over a period of 1 week. We notice that the execution time varies significantly across different invocations of the same application.*

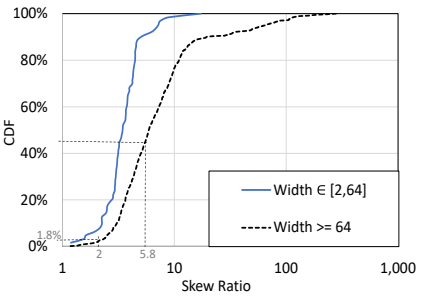


Figure 6: Skew CDF for stages with different width ranges. 98.2% of the DAGs have a skew ≥ 2X, and skew increases for wider DAGs.

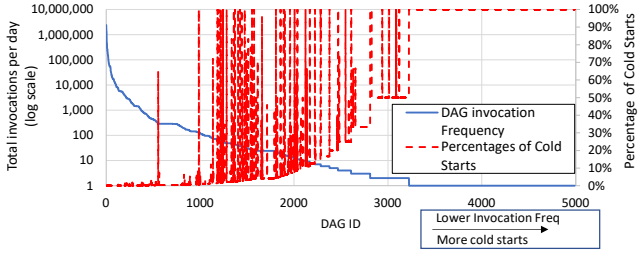


Figure 7: DAG invocation frequency (Blue solid) and its impact on the percentage of cold starts (Red dashed). DAGs with low invocation rate (e.g., once per day) always experience a cold start, whereas very frequent DAGs (e.g., ≥ 500 invocations per day) have very rare cold starts.

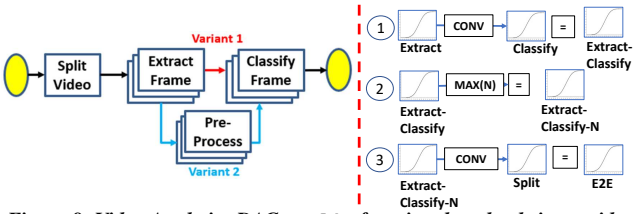


Figure 8: Video Analytics DAG. `Split` *function downloads input video and splits it into chunks. Each chunk is passed to an instance of* `Extract` *function — extracting a representative frame, sent either to* `Classify` *function (Variant #1), or sent to Pre-process function and then* `Classify` *(Variant #2). The steps of estimating E2E latency distribution for Variant #1 are on the right.*

Ignoring correlation by assuming statistical independence leads to over-estimating the combined distribution for in-parallel functions, while it leads to under-estimation for in-series functions. In our evaluation, we give quantitative evidence of these effects (§ 5.4.2) and show that a performance model that is distribution-agnostic (*e.g.*, SONIC [40]), or correlation-agnostic (*e.g.*, [26]), fails to provide accurate E2E latency estimates.

3 Design

We first describe the performance model and E2E latency estimation. Then, we describe how we use the performance model to perform our three optimizations.

3.1 Modeling E2E Latency Distribution

Modeling Functions Runtimes. We represent the runtime of a function as the sum of its initialization and execution times. Since both phases have a high variance, we represent them as separate distributions. This separation allows us to estimate the gains from each optimization. Allocating the right resources and bundling mainly impacts the execution times, whereas pre-warming reduces initialization times.

Combining Latency Distributions. Given a latency distribution for every function, ORION applies a series of statistical operations to estimate the DAG E2E latency distribution.

For two in-series functions with latency distributions represented as random variables X and Y, we use `Convolution` to estimate their combined distribution as: $P(Z=z) = \sum_{\forall k} P(X=k, Y=z-k)$.
If X and Y are independent, we simplify the computation: $P(Z=z) = \sum_{\forall k} P(X=k) \cdot P(Y=z-k)$.

The latter is simpler to estimate since it only requires the marginal distributions of the two components, which can be profiled *separately*, rather than *jointly*.

On the other hand, if the two functions execute in parallel, then their combined latency distribution will be defined by the max of the two. Therefore, we use the `Max` operation to combine their CDFs as follows: $P(Z \le z) = P(X \le z, Y \le z)$. Similar to the `Convolution` operation, a simpler form can be used when X and Y are independent: $P(Z \le z) = P(X \le z) \cdot P(Y \le z)$, which uses the marginal CDFs of the two functions, rather than their joint CDF.

Handling Correlation Among Functions.

We consider two types of statistical correlation in the DAG: in-series and in-parallel correlation. For example, our Video Analytics application (Figure 8) has high correlation between *Pre-process* and *Classify* stages, and also high correlation between parallel *Extract* invocations or parallel *Classify* invocations. (Table 1). By analyzing the correlation between the stages as well as the correlation between the parallel invocations in the same stage, ORION identifies both types of correlation. We consider a Pearson correlation coefficient value > experimental parameter θ, as an indication of correlation. This then determines whether to apply the independent

or dependent formulation for the CONV or the MAX operation. In our experiments, we find that the performance of ORION is relatively insensitive for $\theta \in (0.2, 0.5)$ and we run all the experiments with $\theta = 0.4$.

To determine the length of correlation chains (pairwise, etc.), ORION uses conditional entropy measurements of the execution time and compares the reduction in entropy by including additional terms. Thus, if marginal entropy of stage Y is $H(Y) \gg H(Y|X_i) \approx H(Y|X_i, X_j) \approx H(Y|X_i, X_j, ..., X_s)$, where X_i denotes the random variable of invocation i's execution time, then ORION infers that correlations across stages are at most pairwise. We find that correlations in all our application DAGs *across* stages are at-most pairwise. Our formulation, however, can handle any degree of correlation, not just pairwise. Only the amount of profiling data needed will increase with higher degrees of correlation.

In case of high correlation between N parallel invocations in the same stage, the max operation can be expanded by the chain rule: $P(Z \leq z) = P(X_1 \leq z).P(X_2 \leq z|X_1 \leq z).....P(X_N \leq z|X_1 \leq z, X_2 \leq z, \ldots, X_{N-1} \leq z)$, which is further simplified in case of pairwise correlations by only conditioning on one invocation, hence all conditional terms reduce to: $P(X_i \leq z|X_{i-1} \leq z)$.

Since all components within a stage are identical, we estimate the above equation as follows:

$$P(Z \leq z) = P(X_i \leq z).[P(X_k \leq z|X_i \leq z)]^{n-1} \quad , k \neq i \tag{1}$$

Therefore, we use two distributions to model the max for *any* number of parallel components — the marginal distribution and the conditional distribution.

In practice, all individual components are used to estimate the marginal distribution and all pairs of components are used to estimate the conditional distribution, as all marginal distributions are identical and so are all conditional distributions.

Using this performance model, ORION designs three optimizations for serverless DAGs, which we describe next — ***Right-sizing*** in § 3.2, ***Bundling*** in § 3.3, and ***Right pre-warming*** in § 3.4.

3.2 Allocating the Right Resources

The target of this optimization is to assign the right resources for each function in the DAG so that the entire DAG meets a latency objective with minimum cost. Normally, the user picks the VM-size for each function, and the VM-size controls the amount of allocated CPU, memory, and network bandwidth capacities. What makes this problem challenging is twofold — the scaling of multiple orthogonal resources is coupled together and the scaling of different resources is not linear. As an example, the Classify-Frame function in the Video Analytics DAG has a small memory footprint (540 MB). However, increasing its VM-size above 1,792 MB (as that size comes with a full vCPU [7]) reduces latency. This is because larger sizes come with a fraction of a second core up to six cores, which this function utilizes. The first step in this optimization is to map a given configuration candidate (*i.e.*, a vector of VM-sizes, with one entry per stage) to the corresponding latency distribution. To achieve this, we build a per-function performance model that maps VM-sizes to latency distributions. Next, we combine these distributions to estimate the E2E latency distribution.

Per-function Performance Model. For each function in the DAG, we collect the latency distributions for the following VM sizes: *min* (the minimum VM size needed for this function to execute), 1,024 MB, 1,792 MB, and *max*. We pick 1,024 MB as it is the point of network-bandwidth saturation (increasing VM-size beyond it does not provide more bandwidth), and 1,792 MB as it comes with one full CPU core. Hence, this initial profiling divides the configuration space into 3 regions: [Min, 1024), [1024, 1792), and [1792, Max].

In order to estimate latency distribution for intermediate VM-sizes, we use percentile-wise linear interpolation. For example, the P50 for 1408 MB is interpolated as the average between the P50 for 1,024 MB, and the P50 for 1,792 MB settings. This generates a *prior* distribution for these intermediate VM-size settings. To verify the estimation accuracy in a region, ORION collects a few test points using the midpoint VM-size in that region to measure its actual CDF (*i.e.*, the *posterior* distribution) and compares it with the *prior* distribution. If the error between the *prior* and *posterior* CDFs is high, ORION collects more data for the region midpoint and adds it to its profiling data. In summary, this approach divides the space of a potentially non-linear function into a set of approximately linear functions, and hence, more complex functions get divided into more regions, with a profiling cost overhead. In practice, we find that 5 to 6 regions accurately model the latency distributions for all functions in our applications.

Optimizing Resources for a Latency Objective. Since the performance model estimates the E2E latency distribution of the DAG, we can use it to choose a configuration (*i.e.*, the set of VM sizes) to execute a DAG in order to meet a user-specified latency objective while reducing $ cost. We search for the configurations using a heuristic based on Best-First Search (BFS) [57]. The pseudo-code is shown in Algorithm 1.

The algorithm starts by creating a priority queue, in which all the new states are added. A state here represents a vector of VM sizes, one for each stage. Each new state expands the current state in one dimension (with a step-size of 64 MB) and the start state S_0 has the minimum VM size for every function. The priority is set to be the difference between the target latency and each state's estimated latency multiplied by the $ cost of the new state (lower value means higher priority). Our chosen heuristic is suitable for our problem because latency is a monotonically non-increasing function of resources allocated to a function. The worst-case complexity of BFS is $O(n * log(n))$, where n is the number of states.

3.3 Bundling Parallel Invocations

Stragglers can dominate an application's E2E latency [11, 16, 36, 53]. Here we show how to bundle multiple parallel invocations in one stage within a larger VM, rather than the

Algorithm 1 Best-First algorithm to identify the best VM sizes for functions in a DAG given a user-defined latency objective.

Input: Latency-Percentile=P, Latency-Objective: T_O
Output: Best VM sizes=S_{best}

```
1: ## Initialize priority queue pq, performance model GetLatency, cost
   model GetCost, StepSize = 64 MB
2: ##Set start state S_0 to minimum VM size for every function in DAG
3: pq.insert(s_0)
4: while pq is not empty do
5:     S_next = pq.pop()
6:     ## Create N new states by adding StepSize to each function
7:     ## Set the priority of each state and add to pq
8:     for i = 0-> |S_next| do
9:         S_new = S_next
10:        S_new.VMsize[i] = S_next .VMsize[i] + StepSize
11:        S_new.latency = GetLatency(S_new, P)
12:        S_new.priority = -1 × S_new.latency × GetCost(S_new)
13:        pq.insert(S_new)
14:        ## Check if latency objective is met
15:        if S_new.latency ≤ T_O then
16:            return S_best = S_new
17:        end if
18:    end for
19: end while
20: ## If no explored state meets the latency objective
21: return State S_best with closest latency to T_O
```

current state-of-practice of executing each in a separate VM. This promotes better resource sharing, thus mitigating skew.

To understand how bundling works, consider the example shown in Figure 9 for a stage of 4 parallel worker functions experiencing load imbalance. Specifically, the load for workers #2 and #4 is low and both require only one time step of execution. In contrast, the load for workers #1 and #3 is higher and requires 7 and 3 time steps of execution, respectively. Also, assume that the workers are scalable and can actually make use of additional resources made available. If we execute each worker function on a separate VM, say with 1 core each, the E2E latency is dominated by the slowest worker and the entire stage will take 7 time steps. However, if we bundle the workers together in a single VM with 4 cores, the E2E latency reduces to 3 time steps only. This is because the straggler workers get access to more resources when the lightly loaded workers finish their execution. Notice that the cost remains the same in both cases because they consume 1 core $\times$ 12 time steps or 4 cores $\times$ 3 time steps for the entire stage.

We make a few observations about the applicability of bundling. *First*, bundling is useful in reducing the latency if the execution skew is due to *load imbalance*, which arises from processing bigger partitions of data, or inputs that require more computation. We detect load imbalances due to content by subtracting latency CDF #1 from #2: (#1) When the function is executed multiple times with the same input. (#2) When the function is executed multiple times with different inputs. Moreover, the higher the correlation between workers, the lower the gap between their execution times, and hence, the lower is the benefit from bundling.

Second, for bundling to be useful, the function has to be *scalable* to benefit from the additional resources. We identify a function's scalability using our performance model to estimate the impact on the function's latency CDF when given more resources (§ 3.2). We benefit from the fact that the community has developed many highly scalable libraries, *e.g.*, [10], which are widely used in serverless applications.

Third, our example in Figure 9 assumes there will be *no contention* between bundled workers. However, in practice, we find that this contention can be high, especially for network-bound or IO-bound functions, as these resources do not scale linearly with the VM size. For example, all VM sizes in AWS Lambda get the same disk space of 512 MB and network bandwidth scales only for VM sizes until 1,024 MB.

Based on these three requirements, ORION identifies the best bundle size in two steps. First, ORION identifies functions that experience execution skew and are scalable using the performance model. Second, ORION searches the space of bundle sizes through multiplicative increase (*i.e.*, bundle sizes of 1, 2, 4, etc.). At each step, ORION collects very few profiling runs (we use 10) to capture contention. The search terminates when bundling more workers causes contention and hence increases the E2E latency. ORION strives to spread stragglers across different VMs, by performing a "redistribute" operation if needed, so that each straggler has excess resources to speed up its execution. Since skew often shows up with temporal locality, we spread the parallel functions among the available bundles in a round-robin manner. For example, for the Video Analytics application, load typically varies gradually across consecutive frames.

ORION's security considerations: ORION does not currently bundle functions in different stages for security purposes. Moreover, all the invocations to be bundled together belong to the same user and the same DAG invocation. Additionally, in cases when the stages have very different resource requirements, it becomes counter-productive to come up with one VM size that fits multiple stages. We defer the possibility of bundling invocations across different functions as future work.

3.4 Pre-warming to Mitigate Cold Starts

We describe our approach to mitigating cold starts, leveraging the DAG structure of the application. We describe how to identify when to start pre-warming the VMs for each stage in the DAG, in order to balance the E2E latency and the utilization of the computing resources. This step is performed after we perform the previous two optimizations: Right-sizing and Bundling. Figure 10 shows conceptually the impact of different pre-warming delays on E2E latency and utilization. At the extreme, a delay of zero for every stage minimizes the E2E latency but also minimizes the utilization. The other extreme is no pre-warming at all, which is the state-of-practice. First, we define `pre-warming delay` for a stage S as the time elapsed between the start of the DAG execution and the beginning of initialization of the VMs for that stage. For a given DAG of

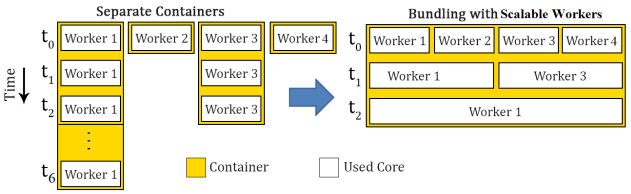


Figure 9: (Left) Separate VMs: workers #2 & #4 finish early, while workers #1 & #3 take longer. (Right) Bundling: after workers #2 & #4 finish, workers #3 & #4 get more resources reducing stage latency.

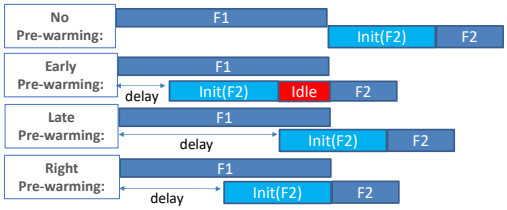


Figure 10: Impact of different pre-warming decisions on the E2E latency and utilization for a chain of two in-series functions. Without pre-warming, the E2E latency increases due to added initialization time of function F2. Both early and late pre-warming are not desirable.

N stages, we want to select a vector $\vec{d} = [d_1, d_2, ...d_N]$ representing the pre-warming delays for each stage in the DAG. For the first stage in the DAG, we have the degenerate case and set its delay (d_1) to zero. This is because pre-warming requires predicting when the DAG will be invoked, which is challenging in the general case. The optimal delay vector, given a performance model $\mathcal{P}$, is defined as follows:

$$\vec{d}^* = \arg\min_{\vec{d}} \; E2E\text{-}Latency(\mathcal{P}, \vec{d}) \quad \text{subject to} \quad Util(\mathcal{P}, \vec{d}) \geq \text{Target Utilization} \tag{2}$$

The selected vector is the one that minimizes the DAG E2E latency while achieving the target resource utilization as set (and dynamically adjusted) by the provider. Both the utilization and the E2E latency are estimated by our performance model $\mathcal{P}$. The metric $Util(\mathcal{P}, \vec{d})$ measures the utilization for a given delay vector using the performance model, and is estimated as $\frac{BusyTime(VM)}{BusyTime(VM)+IdleTime(VM)}$. $BusyTime(VM)$ includes both initialization and execution times, while $IdleTime(VM)$ is the time between when the initialization completes to when the function starts executing. We again use Best-First Search (BFS) to select vector $\vec{d}^*$ as follows. We start by setting all values of $d_i = 0$. In each iteration, we add a delta (100 ms) to the delay factor d_i that yields the best improvement in utilization over the current state without increasing the E2E latency. The algorithm terminates when adding delta to any delay factor does not improve utilization but increases E2E latency.

3.5 Further Design Considerations

Deployment. ORION is designed to serve as a DAG optimization layer. Although the primary use case is to be deployed by the provider, ORION can also be deployed by end-users. In the latter, users are able to select the VM sizes for their functions. For this, the end-user will need to profile her code and also send pre-warming requests to the provider at the right times, as identified by ORION. However, users do not need to change their function code for Bundling. Instead, ORION identifies the bundle size for each parallel stage in the DAG and executes multiple invocations together. The cloud provider still decides the mapping of specific VMs to function bundles.

Naturally, ORION's performance model is trained faster for functions with higher invocation frequency as they provide natural training data points. As discussed in 2.1, frequently invoked DAGs dominate the total set of DAG invocations. For example, the 5%-most frequent DAGs have an invocation rate of 2.3K per day. Hence, it will take us less than 3.5 hours to gather 300 training samples per function. We can accelerate this, and also handle less frequently invoked DAGs, by inserting synthetic but realistic DAG invocations to generate training data points. It is also possible that we will have to re-train our models from time to time when the workload characteristics have changed significantly, or less commonly, the application DAG or the infrastructure characteristics has changed significantly. This incremental training is *not* a computationally heavy task as it involves updating only *parts* of the distribution curves. Maintaining the latency data for performing such updating is also *not* a memory-heavy task.

4 Implementation

We implement ORION in C# and Python 3.8 with 2,100 LOC. We execute the serverless DAG applications in AWS Lambda and use Amazon S3 for data passing between the functions. We use AWS Step Functions [6] to orchestrate the DAG. Function bundling is implemented without any code change by using a wrapper around the (developer-provided) entry point to each function. We use the Python multiprocessing library [9] to execute bundled invocations together.

Runtime Overhead. In theory, the worst-case runtime of Algorithm 1 increases exponentially with the number of stages in the DAG. However, we find that ORION's BFS algorithm has a very low overhead in practice: Each iteration in Algorithm 1 takes [3,7.5] msec, and the best solution takes between 6 and 88 iterations while exploring < 1% of all possible states. The number of iterations depends on the latency target, the steepness of the latency-VM size relation, and the step size (we use 64 MB). For finding the best pre-warming delays, BFS takes between [0.4,3] seconds across all applications.

Scalability. We evaluate the scalability of ORION in Figure 18 with respect to increasing the number of stages. We synthetically replicate the last (and most time consuming) stage of the Video Analytics application to create a DAG of up to 8 stages. The overhead is defined as the inference time divided by the application lifetime, and it ranges between 0.12% for 3 stages to 0.07% for 8 stages. Also, increasing the number of stages, the prediction error increases, but slowly. Specifically, with up to 8 stages, P50 error is stable, and P90 and P95 increase

slowly but never reach 15%. With wider DAGs, our inference time remains unchanged as the fanout degree is used as a parameter in our estimation of MAX (Eq. 1).

Pre-warming. Ideally, implementing pre-warming in AWS Lambda requires our control over assigning invocations to warm containers or VMs. Since we do not have such control, we rely on AWS Lambda's container reuse to implement pre-warming. Specifically, we perform pre-warming by sending a dummy call to a function, then send the actual call right after the response from the dummy call is received.

5 Experimental Evaluation

We evaluate ORION running on AWS Lambda. First, we describe the three serverless DAG applications used in our evaluation. Then, we show an E2E evaluation compared to three alternatives. Next, we show a set of microbenchmarks to evaluate each component of ORION. Finally, we provide a unit experiment on Azure Functions, a platform that allows less configurability for an external mechanism like ORION.

5.1 Serverless DAG Applications

Video Analytics. This application, adopted from Pocket [35], analyzes an input video by extracting representative frames from the video and classifying each frame. The application stages are shown in Figure 8. The first variant of this application directly calls a third function, *Classify-Frame*, which uses a YOLO [45] pre-trained DNN model to classify object(s) in the frame into 1,000 classes. The second variant calls an intermediate pre-processing function, *Pre-process*, which applies a sharpening filter to improve image quality before classification. We refer to this variant as "Video Analytics w/ Preprocess" (VA-Pre, for short). Finally, all classification results are uploaded to remote storage. For VA-Pre, there is a high correlation between the times of the *Pre-process* and the *Classify* functions. We use 600 YouTube videos (300 for profiling, 300 for testing), each of length 1 min, belonging to the "Nature" and "News" categories.

ML Pipeline. This application is a machine learning pipeline (adopted from Cirrus [18]).It consists of three stages: dimensionality reduction (PCA), model training, and testing (*Combine*). The second function, *Train-Model*, runs in parallel and each instance trains a decision tree model using the *LightGBM* Python library [37]. In this stage, a user-specified number of functions is triggered (we use 64 trees in our evaluation), and every function trains a different decision tree. The third function, *Combine*, combines the trained models into a random forest and evaluates its joint accuracy on a held-out test dataset. We use the MNIST [23] database of handwritten digits that has a total of 60K images. We execute the application with 600 runs (300 profiling, 300 test), and in each run, we use 5K images to train the ML model, and 15K for testing.

Chatbot. This application trains a domain-specific Natural Language Understanding (NLU) model, whose task is to identify the accurate "intent" of a user-spoken utterance. We use the Chatbots Intent Recognition Dataset, available on Kaggle [44], which consists of 22 intents and 455 utterances. As before, we evaluate with 300 profiling and 300 test runs. The first lambda in this application parses the dataset and constructs bag-of-words representation for all utterances. Next, a stage of parallel lambdas trains One-vs-Rest classifiers with one lambda per intent. The models are then uploaded to remote storage for real-time intent detection.

The three applications cover important characteristics of serverless DAGs. Specifically, Video Analytics and Chatbot have scatter communication pattern, whereas ML Pipeline has a broadcast pattern. They also cover different fanout degrees (22 for Chatbot, 32 for Video Analytics, 64 for ML Pipeline) and their execution times resemble the average latency of DAGs in our workload characterization (§ 2.1). Moreover, Video Analytics and ML Pipeline are both compute bound, whereas Chatbot is network bound.

5.2 ORION and Competing Approaches

We compare our E2E latency and cost to multiple resource allocation, skew mitigation, and pre-warming approaches:

(1) **Best-Memory:** This is a resource allocation approach that uses our performance model and progressively increases the VM size for every function in the DAG till the latency objective is met. This mimics the standard VM autoscaling that is employed in many cloud scheduling solutions [2, 59]. All invocations run in separate VMs of the same size.

(2) **CherryPick [5]:** CherryPick uses Bayesian-Optimization (BO) to find latency-optimized memory configurations. BO relies on an *acquisition* function to propose new points to sample next. This makes BO a distribution-agnostic baseline as each VM size is profiled once, then a new size is selected by the acquisition function. We set the loss function in BO to be the difference between the achieved latency and the user-specified latency target. Since CherryPick is distribution agnostic, it cannot detect execution time skew and hence performs no bundling. We choose CherryPick as it (BO with Gaussian processes) was recently demonstrated as the most competitive approach in the category [15] and recent approaches have used it for configuring serverless functions [3].

(3) **Speculative-Execution:** This is a skew mitigation approach that identifies stragglers at runtime and executes a duplicate invocation on a different VM. Speculative execution is widely used in MapReduce, Hadoop, and Spark systems for skew mitigation to reduce tail latency [11, 19, 31, 51]. We adapt this baseline from Spock [30] (specifically the technique called "conservative autoscaling in predictive mode"). Since the skew is caused by the input's content, the new invocation will likely take as long as the first one unless it is assigned more resources. Accordingly, we modify the technique by assigning the "Max" resources (10 GB) for the new invocation.

(4) ORION **Right-Sizing:** This variant of ORION performs Right-Sizing only, and not the other two optimizations.

(5) ORION **Full:** This is our complete solution, which includes

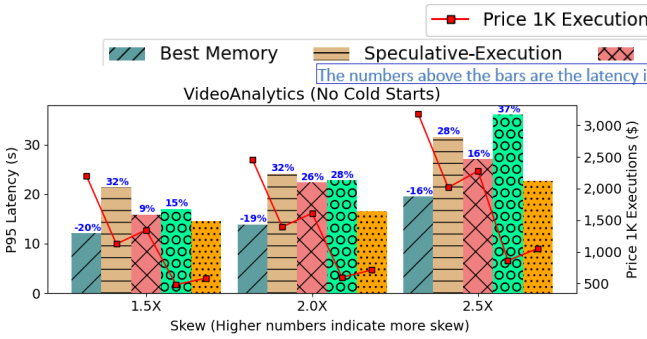


Figure 11: Skew is varied by changing the detection probability threshold as: 2%, 10%, and 15%, with lower values resulting in more detected objects and higher skews [22].

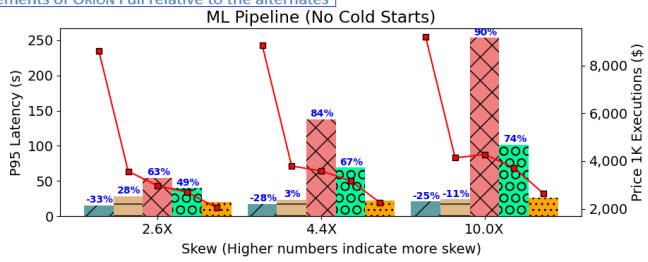


Figure 12: Skew is varied by changing the maximum value for each hyper-parameter, e.g., we vary the max number of trees as: 50, 100, and 200, and these map to 2.6×, 4.4×, and 10× skews.

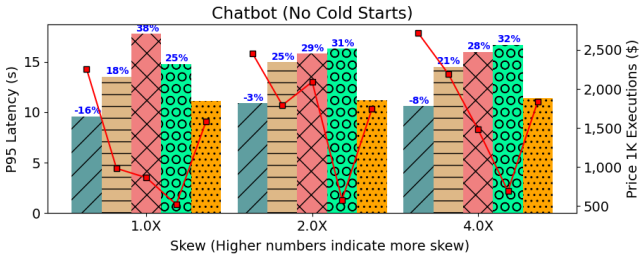


Figure 13: Skew is altered to 1×, 2×, and 4× by changing the number of training epochs as: 100, 500, and 200.

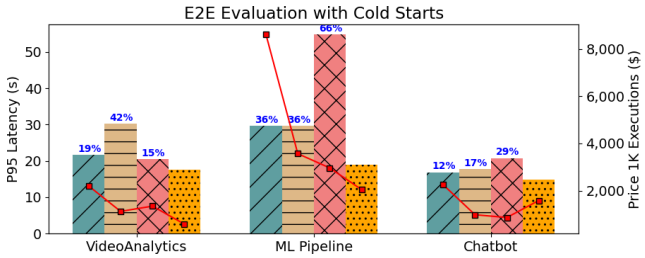


Figure 14: E2E evaluation with cold starts. ORION *achieves the lowest latency and cost compared to all baselines.*

Right-Sizing, Bundling, and Pre-Warming.

5.3 End-to-End Evaluation

We show the P95 latency (primary Y-axis, shown using bars) and cost (secondary Y-axis, shown using lines) of each approach in Figures 11, 12, and 13 for the three applications. The numbers above the bars are the latency improvements of ORION Full relative to the alternates. First, we set the latency objective to the minimum achievable latency, which is identified by executing all functions with *max* VM size, while computation skew is minimized. For each application, we vary the skew in a controlled manner through application-specific parameters. For each solution and for each experimental point, we execute each application 300 times and highlight the gains of ORION's right sizing and bundling. In this part, we take care to eliminate all cold starts for the experimental data points. Later, we show the impact of cold starts and the additional gain due to ORION's pre-warming design in Figure 14. Compared to Best-Memory, ORION has a slightly higher latency since Best-Memory assigns high resources to all workers, including stragglers. However, this baseline increases the cost significantly by assigning identical resources for each stage and parallel running workers in separate VMs, which over-provisions the resources to meet the latency objective. ORION provides [33%,71%] lower cost by assigning the right resources for each function and bundling parallel workers. Compared to Speculative-Execution, we notice that ORION has consistently lower latency and cost across all skews. For example, with the lowest skew, ORION shows [18%, 32%] lower latency and [46%, 57%] lower cost for the three applications. This is because Speculative-Execution detects straggling workers (using a user-specified threshold) and re-executes them on new VMs with the max size. This causes an additional delay due to the wasted execution time. It also increases cost as it sometimes mistakenly re-executes workers that would finish shortly after the threshold. ORION's bundling does not require any user-specified threshold to detect straggling workers, assigning them more resources once co-located workers finish and release their resources.

For CherryPick, since it is distribution-agnostic, we modify the BO algorithm so that 100 points are profiled for each point selected by BO's acquisition function to measure the latency percentiles. We run CherryPick for 100 iterations total (a generously high number compared to the original work and follow-on works), resulting in 10K profiles for each application. Notice that ORION requires only 300 profiling runs to model the E2E latency distribution, reducing the profiling burden of CherryPick by 97%. Compared to CherryPick, we notice that ORION Full consistently provides lower latency and cost, except for Chatbot where CherryPick has higher latency but lower cost. Specifically, with the highest skew, ORION Full shows [16%, 90%] lower latency and [38%, 53%] lower cost for Video Analytics and ML Pipeline. For Chatbot, this application has a lower bundle size than the others, reducing the gain from ORION's bundling mechanism. Compared to ORION Right-Sizing, adding bundling significantly reduces the latency across the three applications. However, bundling causes a slight increase in the cost for Video Analytics by

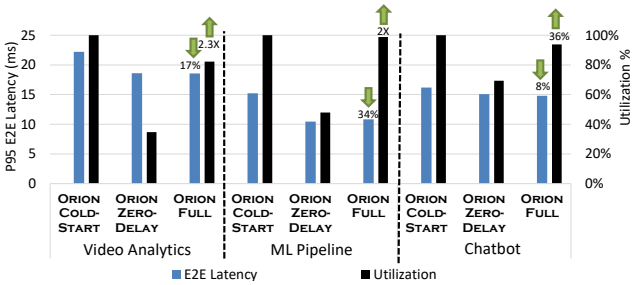


Figure 15: Impact of pre-warming on latency and utilization. We use VM and bundle sizes selected by* ORION *and compare different execution strategies w.r.t. cold starts. Percentages over the bars of* ORION *show the improvements in P95 Latency (over* ORION *Cold-Start) and Utilization (over* ORION *Zero-Delay pre-warming).

22% (relative to No-Bundling), while it causes a decrease in cost for ML Pipeline by 30%. The reason is that the ML Pipeline experiences higher skews (up to 10X), and for higher skew, Bundling is more beneficial. We also notice that the reduction in latency increases with higher skews. For Chatbot also, bundling reduces the latency compared to no bundling, but increases the cost by 163%. This is because the Chatbot application is more network bound and not compute bound than the other two, and hence the best bundle size is only 2, vs [6,8] for the other applications. However, cost with bundling is still 33% lower than Best-Memory, which is the closest to us in latency among all baselines.

Mitigating Cold Starts with Pre-warming. So far, we have compared ORION to the baselines with only warm executions. Now we show the gain of our pre-warming technique and how useful it is in reducing cold starts. Figure 14 shows ORION's latency and cost vs other baselines in the case of cold start for every function in the DAG. We notice that all baselines are impacted by cold starts and their latencies increase, whereas ORION's pre-warming technique is able to mitigate the impact of cold starts. For example, Best-Memory shows an increase of E2E latency over ORION by 19%, 36%, and 12% for Video Analytics, ML Pipeline, and Chatbot, respectively. Similarly, Speculative-Execution suffers from cold starts twice, once for the first execution with the small VM, and once more for the second execution with the max VM size. Hence, ORION's improvements in latency over Speculative-Execution increase to 42%, 36%, and 17% for the three applications. To summarize, ORION's three optimizations of Right-sizing, Bundling, and Right pre-warming provide lower E2E latency and cost over all competing approaches. In the next section, we show a set of microbenchmarks to separately evaluate the performance of each component of ORION.

5.4 Microbenchmarks

5.4.1 Impact of Pre-warming on Utilization & Latency

Figure 15 shows the latency and utilization achieved by ORION versus its two variants. The first, called ORION Cold-Start, does not perform any pre-warming and hence suffers from increased latency, yet has very high utilization as it causes no idle times. The second, called ORION Zero-Delay, initializes all the containers with zero delay for all stages, *i.e.*, at the beginning of the DAG execution. Hence it ensures the lowest latency that can be achieved, but incurs increased idle times due to early pre-warming and hence suffers from low utilization. On the other hand, ORION Full uses the right delay times identified by BFS (§ 3.4). As shown in Figure 15, ORION Full consistently achieves lower latency than ORION Cold-Start, and consistently higher utilization over ORION Zero-Delay for all three applications. We also notice that the latency gains are higher for ML Pipeline and Video Analytics than for Chatbot, which is due to the higher initialization times observed in these two applications when downloading the heavy ML packages and the large pre-trained object detection models. Therefore, estimating the right values of the delays for each stage, as done by ORION, is essential to mitigate cold starts without significantly reducing utilization.

5.4.2 Evaluation of Performance Model

Capturing Correlation between Functions.

Here we evaluate how much correlation exists in our target applications. We calculate the Pearson's correlation coefficient between in-series functions (*e.g.*, between *Split-Video* and *Extract-Frame*), and between in-parallel functions (*e.g.*, between multiple instances of *Extract-Frame*). We show the correlation scores in Table 1 for Video Analytics.

Table 1: **Correlation between execution times of functions in the Video Analytics DAG. In-series correlation is low but in-parallel correlation is high.**

VM-Sizes (in MB)	In-series Correlation			In-parallel Correlation	
Split, Extract, Classify	Split ⇕ Extract	Extract ⇕ Classify	Preprocess ⇕ Classify (VA-Pre)	Extract	Classify
192, 192, 576	0.09	0.04	0.45	0.05	0.43
1024, 1024, 1024	0.07	0.02	0.61	0.34	0.44
1792, 1792, 1792	-0.07	-0.04	0.69	0.48	0.58
3008, 3008, 3008	0.05	-0.01	0.88	0.65	0.51

The correlation scores between in-series components are close to zero (0.036 on average for Video Analytics, 0.06 for ML Pipeline, and 0.04 for Chatbot), while the correlation scores between functions in the same stage are high for Video Analytics (0.55), while low for ML Pipeline (0.052) and for Chatbot (0.03). For Video Analytics w/ Preprocess, `Pre-process` has a high correlation with in-series `Classify` functions (0.65). Therefore, we apply the dependent `conv` operation between `Pre-process` and `Classify`, while we use the independent `conv` operation for all other in-series functions for all applications. Additionally, we incorporate correlation when performing `max` operation (if correlation is detected) by using the conditional distribution.

Estimating E2E Latency Distribution. Here we evaluate the accuracy of ORION in predicting the E2E latency

Table 2: **Video Analytics: Error rates for ORION's E2E latency estimation. Abbreviations: S→Split, E→ Extract, and C→Classify**

Video Analytics						
VM Sizes (MB)	ORION		Distribution Agnostic		Correlation Agnostic [26]	
S, E, C	**P50**	**P95**	**P50**	**P95**	**P50**	**P95**
512 , 1280 , 1536	14.0%	13.0%	40.0%	15.6%	78.7%	47.5%
768 , 1280 , 2240	14.0%	12.0%	35.4%	11.6%	67.7%	38.3%
1536 , 512 , 1536	13.0%	11.0%	39.7%	16.7%	79.4%	49.9%
1792 , 1792 , 576	6.4%	11.8%	11.6%	-39.0%	49.7%	-18.2%
6000, 6000, 6000	14.5%	10.7%	24.2%	3.3%	56.9%	30.5%
MAPE	**13.0%**	**12.0%**	**32.0%**	**21.0%**	**68.0%**	**39.0%**

distribution for the entire DAG. We compare to two baselines — distribution-agnostic (as mentioned earlier, any BO-based technique like CherryPick falls in this category) and correlation-agnostic (*e.g.*, [26]). The results are shown in Table 2 for Video Analytics.

ORION estimates the E2E latency distribution for the applications with low error rate (<15%), much lower than those of both baselines. We find through drill down of our estimation error that: (i) our estimated length of correlation chains as pairwise (§ 3.1) is accurate and hence does not lead to much error (ii) the dominant source of error lies in the interpolation of the CDFs for each function for the *unseen* memory configurations. This is despite our design, where if the interpolation causes too much error, the memory region is split into two and further data points are collected (§ 3.2). These observations hold across all three applications. Error rates are higher in Video Analytics relative to ML Pipeline because the execution time is content sensitive for the former. Our technique does *not* create content-specific models since we (and any provider-side tool) cannot have visibility into user data due to privacy concerns. The ***Distribution-Agnostic*** baseline uses the median execution times and predicts the median execution times for unseen configurations by interpolation. This baseline has a high error rate in the range of [-39%, 40%] for Video Analytics, [-5%,108%] for ML Pipeline, and [-6%, 66%] for Chatbot. The ***Correlation-Agnostic*** baseline from [26] also has a higher error rate in the range of [-18%, 79%] for Video Analytics, [-5.4%,103%] for ML Pipeline, and [80%, 111%] for Chatbot. Note that the majority of the errors for the ***Correlation-Agnostic*** baseline are over-estimation, which is caused by ignoring the correlation between parallel workers. In conclusion, it is important to take into account the latency distributions and not simply a point estimate and to account for the correlation across stages and across workers within a stage, even when the correlations are quite weak (Table 1).

5.4.3 Optimizing Resources for a Target E2E Latency

We profile the applications to build the E2E performance model in ORION for all three applications as mentioned in § 5.4.2, then set 6 latency targets per application. ORION proposes the DAG configuration (*i.e.*, VM size for each function in the DAG) to meet each latency target at while reducing cost. We validate ORION's accuracy by executing the application with the proposed configuration and comparing the achieved latency to the user requirement.We notice that ORION's proposed configurations are very close to the latency requirement in all applications, with error rate of [-2.75%, 4.93%] for Video Analytics, [-1.37%, 2.6%] for ML Pipeline, and [-3.5%, 3.7%] for Chatbot. Table 3 lists detailed configurations for Video Analytics. We notice that expectedly, ORION tends to assign more resources as the latency percentile increases (*i.e.*, P50 → P95) or as the latency requirement decreases (50 sec → 30 sec). Also ORION decides to increase the allocated resources for a subset of functions and by different amounts, based on the latency requirement. For example, for ML Pipeline, ORION increases the VM-size of PCA from 768 MB to 832 MB to achieve a latency requirement of (P90 $\leq$ 50 sec). However, ORION decides to increase the VM-size of Combine from 1,408 MB to 1,472 MB to achieve a latency requirement of (P90 $\leq$ 40 sec). This shows ORION's BFS adjusts the *Best* function to increase its resources according to the estimated latency of the current state.

5.4.4 Impact of Varying Bundle Size

We evaluate the impact of varying bundle sizes on the E2E latency CDF and cost (Figure 16). First, we run our Video Analytics application with the best VM size selected by BFS but without bundling. This is an application that is both CPU bound and scalable, and thus a good candidate for demonstrating the effect of bundling. Next, we progressively increase the bundle size and the VM size proportionally. For example, if the best VM size selected by BFS is 1,792 MB (1 core), we use a VM of size 1,792 $\times$ 2 when we bundle pairs together, and so on. We notice that increasing the bundle size from 2 to 6 workers reduces the latency; however, increasing the bundle size beyond that (to 10 and 30) causes an increase in the latency. This is because the maximum number of cores available in AWS Lambda is 6 and hence, at the higher bundle sizes (10 or 30), each worker is getting less than its required resource.

Thus, the design of ORION to choose the best bundle size is essential to optimize latency by avoiding contention.

5.5 Generalizability to Microsoft Azure

To test if ORION generalizes to other FaaS providers, we evaluate our model using Azure Functions as the serverless environment. Azure Functions supports a few plans, but the most popular one is the *Consumption Plan*. In this plan, users are charged for the exact amount of resources consumed by their functions at runtime, whereas all other plans have a flat rate pricing model. Although we have no control over the resources assigned to individual functions when selecting the *Consumption Plan*, we wanted to measure the accuracy of ORION's E2E latency estimates compared to the actual la-

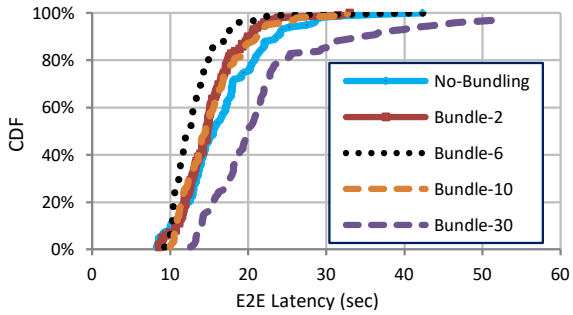


Figure 16: Video Analytics: Impact of varying bundle sizes. No-bunling has high latency due to computation skew. The optimal bundle size here is 6, and using a bundle size of $\geq$ 10 causes contention and the latency increases.

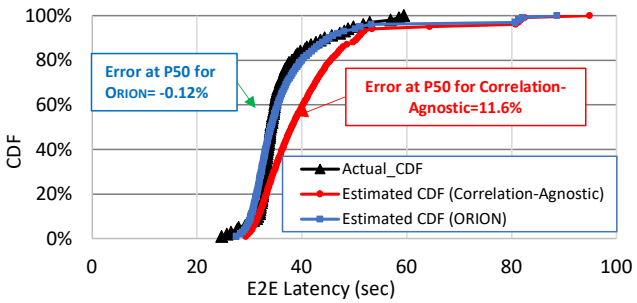


Figure 17: ORION's estimated latency CDF vs Actual CDF for Video Analytics application deployed in Azure Functions. Ignoring in-parallel correlation leads to higher errors for the Correlation-Agnostic baseline.

Table 3: **ORION's E2E latency-optimized VM sizes. ORION meets the latency objective with a low error rate in the range of [-2.75%, 4.93%]**

Video Analytics			
User Requirement	ORION's configs (MB) Split,Extract,Classify	Achieved Latency	Error Rate
P50 $\leq$18 s	192, 192, 640	18.3 s	1.5%
P95 $\leq$18 s	384, 192, 768	17.5 s	-2.8%
P50 $\leq$17.5 s	192, 192, 704	18.4 s	4.9%
P95 $\leq$17.5 s	640, 192, 768	17.4 s	-0.5%
P50 $\leq$17 s	256, 192, 768	17.8 s	4.4%
P95 $\leq$17 s	832, 256, 1024	17.3 s	2.0%

tency observed with this plan. We show ORION's estimated CDF and actual CDF in Figure 17. We use our Video Analytics application with the earlier-mentioned 600 YouTube videos.

We use our E2E performance model to estimate the CDF for the entire DAG. For fair comparison to AWS-Lambda, we also rely on remote-storage (*i.e.*, Azure Blob Storage) for data-passing between the functions. We also show the estimated CDF when correlations among functions are ignored — this corresponds to the "Correlation-Agnostic" baseline from our earlier experiment (§ 5.4.2). We notice that ORION predicts the E2E latency with very low error rates (-0.12% for P50, 1.9% for P90, and 2.5% for P95 latencies). The Correlation-Agnostic baseline has significantly higher errors (11.6% for P50, 14.4% for P90, and 29.2% for P95). Thus, the baseline suffers more for higher percentiles.

6 Pre-warming Policy Simulator

To better understand different pre-warming policies without being constrained by privileges granted by the cloud provider, we build a policy simulator, implemented in Python 3.8 with 1,058 LOC. The simulator takes as input the latency CDFs for stages in the DAG. Policies are implemented through a state machine with different actions being taken in each state (such as FUNC_START, FUNC_END, FUNC_PREWARM, etc.). The output of the simulator are the E2E latency CDF of the DAG and the overall resource utilization. We open source the simulator for future exploration of serverless DAGs [1].

Simulation Results. Figure 19 shows the utilization achieved by a policy with optimal pre-warming using an Oracle that knows the exact runtimes of each function invocation. The input DAG has 2 stages with width of 10 for each stage. The X-axis denotes the skew on the runtime of the first stage. The Y-axis denotes the percentage of variance on the delay chosen by the Oracle for pre-warming functions of the second stage — so if the value is $X\%$ and ORION calculated deterministic delay is Y, then the Oracle can pick a delay in the range $[Y - X\%$ of $Y, Y + X\%$ of $Y]$. Thus, the range of values the Oracle can choose from is capped even if the Oracle determines the optimal pre-warming time for a specific function invocation lies outside of the range. The lowest point on the Y-axis is the optimal deterministic delay determined by ORION for all function invocations in the second stage. We find that the E2E latency is unaffected (not shown) by increasing the size of the range on higher skews, but utilization increases. This is because the policy is able to pre-warm with the ideal delay and hence does not incur any idle time. This shows the theoretical best achievable utilization since we use an Oracle. However, implementing this Oracle has two challenges: (1) Predicting per-function *exact* latency is impractical. (2) Selecting a delay factor for each function invocation rather than each stage increases search space exponentially with DAG width.

7 Related Work

Minimizing cost and/or execution time for serverless chains is the target of a few recent studies. For example, Sequoia [52] makes the observation that current serverless platforms treat functions within a DAG separately, without making use of the DAG structure. SONIC [40] reduces the communication latency between in-series serverless functions by optimizing the data passing strategy. SONIC selects from among the data passing strategies: direct passing, remote storage, and local VM-storage, where only the latter two can be implemented directly in AWS Lambda. Caerus [60] stresses the importance of optimizing latency and cost jointly for serverless DAGs, and achieves this by identifying pipeline-amenable data de-

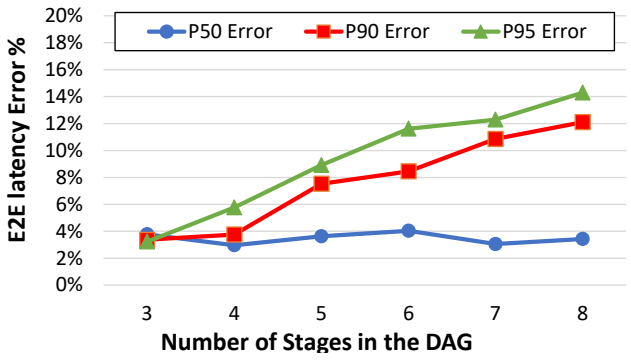


Figure 18: ORION*'s error with varying number of stages. More stages increase the error for the tail, while the median stays stable.*

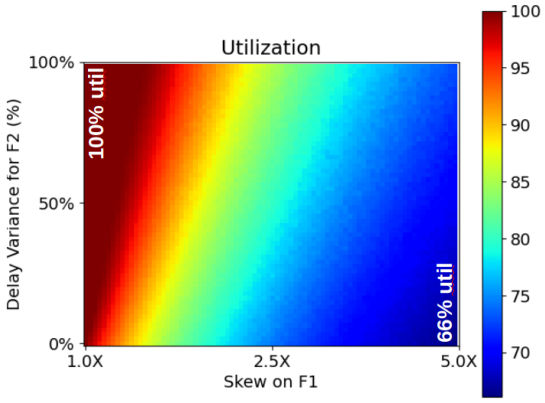


Figure 19: Simulation of an Oracle pre-warming policy where utilization improves with the width of distribution from which the pre-warming delays are chosen. ORION*'s strategy corresponds to the 0% variability,* **i.e.,** *deterministic delay.*

pendencies between stages to find ideal task launch times. Xanadu [21] and Kraken [14] tackle the problem of cascading cold starts in a dynamic DAG. Neither can determine the optimal pre-warming time to mitigate cold starts.

Overall, no prior work in this category considers execution time variance and its impact on cost or utilization.

Latency and Cost Prediction for Serverless Functions. A few prior studies have targeted predicting the execution time and cost for serverless functions. For example, [25] predicts (a point estimate) and optimizes resources for a *single* serverless function by building regression models from a host of synthetic functions. The authors in [26] also observe a variance in execution time in serverless environments, and hence, apply mixture density networks to predict the distribution of the function cost. However, their Monte-Carlo simulation mechanism is very sample inefficient.

ORION uses a more direct method by applying statistical operations to combine the distributions of individual functions and thus, to infer the E2E latency distribution. A number of prior works target reducing the cost of serverless DAGs by optimizing the intermediate data transfer between functions, such as, Costless [27], SONIC [40], Locus [43], and Pocket [35]. They solve an orthogonal problem to ours, namely, reducing the cost of intermediate data transfer. ORION does *not* introduce a new mechanism for intermediate data transfer, nor does it limit or specify the method for state transfer between functions. We use state-of-practice remote storages, such as AWS S3 and Azure Blob Storage. However, ORION would integrate seamlessly with the mentioned systems as the read/write times are included in the latency profiles used in ORION 's model.

Scheduling in Serverless Computing. Photon [24] optimizes *single-stage* serverless functions by doing the equivalent of bundling in ORION, but not for skew mitigation. Its main motivation is to reduce the memory footprint of parallel invocations of a function, while its design sophistication is meant to address security concerns of bundling (out of scope for ORION). One work that targets meeting latency SLAs for serverless DAGs is Atoll [50].

It takes a complementary approach to ours—partitioning a cluster to lower scheduling overheads, and proactively starting up containers and then routing function requests to the appropriate containers.

Resource Optimization in the Cloud. Black-box configuration tuning systems such as CherryPick [5], Selecta [34], OptimusCloud [39], and Ernest [54] target optimizing the cloud resources for a wide range of applications by selecting the right VM type and size, which vary in the amount of allocated resources. However, these systems treat the application as a single component, and thus, do not take the DAG workflow information into account. Further, they are not directly applicable to serverless applications.

Cold Starts Mitigation. Many prior works identified cold starts as a major performance bottleneck in FaaS platforms. Accordingly, several solutions have been proposed such as keeping containers alive [29], leveraging checkpoint/restore operations [49], or using *Pause* containers [41]. Although these solutions reduce the initialization time significantly, there is still a significant user-observable initialization time. ORION hides this initialization time through pre-warming and decides the right time to start pre-warming to minimize idle time, hence keeps resource utilization high.

8 Discussion

Profiling and Modeling Overheads. ORION requires monitoring the execution of the application for a number of runs to accurately capture the latency distribution for each function in the DAG. In our evaluation, for all the applications, a total of 300 profiling runs was found sufficient for accurate 95-percentile latency estimates. Initially, and before convergence is reached, the data collection is performed as a background

process while the DAG executes with user-provided configurations. An important design consideration for ORION is that this data collection does not have to happen purely offline and in batch mode. Rather, that is complemented with online data collection and incremental model refinement, which is a lightweight task. When predicted and observed latencies differ significantly (as can happen if the workload or the application changes), we restart the data collection phase to capture the changes in the latency distributions.

Bundling and Performance Model Interaction. Bundling changes the DAG structure (by reducing the fanout degree), and hence, the performance prediction model needs to be updated. Therefore, this becomes an iterative process, with each iteration being *Performance model building* $\Rightarrow$ *Resource optimization* $\Rightarrow$ *Bundling*. In practice, we find that a single iteration, or at most two iterations, leads to convergence.

Impact of The Three Optimization. The three optimizations of ORION can have a negative impact on performance, resource utilization, or $ cost if not performed carefully. First, over-provisioning the VM size for all workers to mitigate execution skew (as done by the Best Memory baseline in our evaluation) unnecessarily increases the $ cost (Figures 11, 12,& 13). Second, excessive Bundling (bundle size > right bundle size) can lead to resource contention and increase of the latency (Figure 16). Third, early pre-warming (delay < right delay) decreases resource utilization, whereas late pre-warming increases latency (Figure 15). This motivates the need for an accurate performance model to accurately perform these three optimizations. In terms of cost, we notice that users do not pay for initialization times, hence pre-warming does not impact cost. However, the provider should treat a pre-warming request as a hint since a true invocation is always more important.

Applicability of Performance Model.

ORION is tailored to model the performance for serverless DAGs. In general, the response time of a job includes queuing and execution times. Cloud providers operate large serverless platforms, providing virtually infinite capacity, reducing queuing delays to primarily cold-start latencies [38]. Further, serverless platforms typically limit the execution time of each invocation [8] favoring modular reusable functions. The combination of short queuing and execution times enables ORION to model E2E latency, without the need to predict variable (and long), heavy-tailed queuing times that appear in other environments [20, 33, 46].

Mitigating Infrastructure-caused Delays. In serverless platforms, two types of stragglers can be observed: (1) Stragglers that experience longer execution times due to their input content (e.g., larger data portions or more complex inputs such as video frames with many objects). (2) Stragglers that appear due to infrastructure causes (e.g., network fluctuations). Bundling mitigates the first type of stragglers. The second type is well studied in the literature, and solutions such as *Speculative Execution* [11] work well in practice. Nevertheless, Bundling has a positive side effect of using fewer VMs/containers, reducing the likelihood of occurrence for infrastructure stragglers.

Supporting Dynamic DAGs. In a dynamic DAG, the execution flow is identified at runtime, say based on input data. Such DAGs appear in microservice-based applications [14], among others. ORION, as well as other provider-side tools, cannot have visibility into user data due to privacy concerns. Hence, ORION cannot support dynamic DAGs where the path is determined based on request content.

Future Work. Our bundling approach increases VM size proportionally with the bundle size. For example, assuming a single function invocation use a VM of size VM_{single}, we bundle N invocations in a VM with a size of $N \times VM_{single}$. There is, however, room to explore choosing other VM sizes beyond linear scaling. Furthermore, combining two or more in-series functions together to execute in a single VM can improve performance compared to invoking those function in separate VMs (e.g. due to avoiding remote storage communication). We plan to explore the performance benefits of these ideas.

9 Conclusion

We proposed ORION as a novel optimization technique for serverless DAGs. It presents four design innovations: a distribution and correlation-aware performance model for E2E latency, a resource optimization strategy, a design for bundling multiple invocations of a function within a stage to mitigate execution time skews, and a pre-warming strategy to mitigate cold starts. We evaluate ORION on AWS Lambda on three serverless applications with different DAG structures, skews in execution time, and communication patterns. We compare ORION to three competing approaches and show significant improvements in E2E latency, $ cost, or both. We highlight the following insights: (1) It is challenging to decide on the right resource configurations that accurately meet latency SLOs for serverless DAGs. (2) It is important to bundle parallel workers together to mitigate skew, yet it is challenging to pick the right bundle size that avoids resource contention. (3) We can leverage the DAG structure information along with latency CDF estimates to find efficient pre-warming delays that minimize E2E latency without degrading utilization.

10 Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Numbers CCF-1919197, CNS-2016704, CNS-2038986, CNS-2038566, CNS-2146449 (NSF CAREER award), NIH Grant R01AI123037, and funding from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors. The authors thank the reviewers and artifact evaluators for their enthusiastic comments, and the shepherd, Timothy Wood, for his insightful feedback.

References

[1] Pre-warming Policy Simulator. https://github.com/icanforce/Orion-OSDI22, Last retrieved: May, 2022.

[2] Muhammad Abdullah, Waheed Iqbal, Josep Lluis Berral, Jorda Polo, and David Carrera. Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing*, 2020.

[3] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. Cose: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 129–138. IEEE, 2020.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 923–935, 2018.

[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 469–482, 2017.

[6] Amazon. Aws step functions documentation. https://docs.aws.amazon.com/step-functions/index.html, Last retrieved: May, 2022.

[7] Amazon. Configuring lambda function memory. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.htmll, Last retrieved: May, 2022.

[8] Amazon. Lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, Last retrieved: May, 2022.

[9] Amazon. Parallel processing in python with aws lambda. https://aws.amazon.com/blogs/compute/parallel-processing-in-python-with-aws-lambda/, Last retrieved: May, 2022.

[10] Amazon Web Services. Parallel Processing in Python with AWS Lambda. https://aws.amazon.com/blogs/compute/parallel-processing-in-python-with-aws-lambda/, Last retrieved: May, 2022.

[11] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming stragglers in approximation analytics. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 289–302, Seattle, WA, April 2014. USENIX Association.

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

[13] Azure. Durable functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp, Last retrieved: May, 2022.

[14] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 153–167, 2021.

[15] Muhammad Bilal, Marco Serafini, Marco Canini, and Rodrigo Rodrigues. Do the best cloud configurations grow on trees? an experimental evaluation of black box algorithms for optimizing cloud workloads. *Proceedings of the VLDB Endowment*, 13(12):2563–2575, 2020.

[16] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[17] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, volume 2018, 2018.

[18] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[19] Qi Chen, Jinyu Yao, and Zhen Xiao. Libra: Lightweight data skew mitigation in mapreduce. *IEEE Transactions on parallel and distributed systems*, 26(9):2520–2533, 2014.

[20] Mark E Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection scheduling in web servers. Technical report, Boston University Computer Science Department, 1999.

[21] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, pages 356–370, 2020.

[22] DeepQuest-AI. Imageai : Video object detection, tracking and analysis. https://github.com/OlafenwaMoses/ImageAI/blob/master/imageai/Detection/VIDEO.md, Last retrieved: May, 2022.

[23] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[24] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[25] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*, pages 248–259, 2021.

[26] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 265–276, 2020.

[27] Tarek Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312. IEEE, 2018.

[28] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 334–341. IEEE, 2018.

[29] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[30] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 199–208. IEEE, 2019.

[31] Yanfei Guo, Jia Rao, Changjun Jiang, and Xiaobo Zhou. Moving hadoop into the cloud with flexible slot management and speculative execution. *IEEE Transactions on Parallel and Distributed systems*, 28(3):798–812, 2016.

[32] Vipul Gupta, Swanand Kadhe, Thomas Courtade, Michael W. Mahoney, and Kannan Ramchandran. Oversketched newton: Fast convex optimization for serverless systems. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 288–297, 2020.

[33] Mor Harchol-Balter, Mark E. Crovella, and Cristina D. Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing*, 59(2):204–228, 1999.

[34] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, 2018.

[35] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 427–444, 2018.

[36] YongChul Kwon, Kai Ren, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Managing skew in hadoop. *IEEE Data Eng. Bull.*, 36(1):24–33, 2013.

[37] LightGBM. Lightgbm python-package. https://lightgbm.readthedocs.io/en/latest/Python-Intro.html, Last retrieved: May, 2022.

[38] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, 2011. Special Issue: Performance 2011.

[39] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 189–203, 2020.

[40] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301, 2021.

[41] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[42] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 115–130, 2020.

[43] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 193–206, 2019.

[44] python. Chatbots: Intent recognition dataset. https://www.kaggle.com/elvinagammed/chatbots-intent-recognition-dataset, Last retrieved: May, 2022.

[45] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[46] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Internet Technol.*, 6(1):20–52, February 2006.

[47] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 205–218, 2020.

[48] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 281–295, 2020.

[49] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, pages 1–13, 2020.

[50] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 138–152, 2021.

[51] Spark. Spark speculation. https://spark.apache.org/docs/latest/configuration.html, Last retrieved: May, 2022.

[52] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 311–327, 2020.

[53] Jason Teoh, Muhammad Ali Gulzar, Guoqing Harry Xu, and Miryung Kim. Perfdebug: Performance debugging of computation skew in dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 465–476, 2019.

[54] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for Large-Scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, March 2016. USENIX Association.

[55] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1288–1296. IEEE, 2019.

[56] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 133–146, 2018.

[57] Wikipedia. Best-first search. https://en.wikipedia.org/wiki/Best-first_search, Last retrieved: May, 2022.

[58] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[59] Fan Zhang, Xuxin Tang, Xiu Li, Samee U Khan, and Zhijiang Li. Quantifying cloud elasticity with container-based autoscaling. *Future Generation Computer Systems*, 98:672–681, 2019.

[60] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: Nimble task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 653–669, 2021.

[61] Qingyang Zhang, Hui Sun, Xiaopei Wu, and Hong Zhong. Edge video analytics for public safety: A review. *Proceedings of the IEEE*, 107(8):1675–1696, 2019.

A Artifact Appendix

Abstract

This artifact appendix includes all the necessary information to reproduce the main evaluation results of the OSDI' 22 paper: ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs.

Scope and Usage

ORION is a serverless DAG optimization layer implemented in C# and Python 3.8. ORION accepts a DAG as an input and profiles the execution time for each function in the DAG as well as the entire DAG. The execution times are represented as distributions (CDFs) to capture the variability in runtimes. Afterward, users provide ORION with requirements such as a latency target (*e.g.*, P95 $\leq$ 20 seconds) and/or an upper limit on the budget (*e.g.*, cost of 1K executions $\leq$ $1000). Next, ORION performs three optimizations to achieve user-provided requirements. The three optimizations are: (1) **Right-sizing**: Finding the best resource configurations for each function to meet the E2E latency objective with the minimum cost. (2) **Bundling**: Identifying stages where co-locating multiple parallel instances of a function together to be executed on one VM will be beneficial. The benefit arises when there is computation skew among the parallel workers caused by different content inputs and functions are scalable. (3) **Right pre-warming**: The VMs to execute the functions in the DAG are pre-warmed just right, ahead of time, so that cold starts can be avoided while keeping provider-side utilization of resources high. With these three optimizations, ORION accurately meets latency service level objectives (SLOs) while reducing execution cost. The output of ORION is a transformed DAG that has the same semantics as the user-provided DAG, but with higher performance (*i.e.*, lower latency) and lower execution cost.

Contents

1. **Benchmarks-AWS-Lambda:** This folder contains the code for the three evaluation applications (Video-Analytics, ML-Pipeline, and NLP-ChatBot). By running `deploy_application.sh` in each application directory, a DAG serverless workflow can be deployed on AWS Lambda using AWS Step Functions.

2. **DAG_Profile:** This folder contains the code for our DAG profiler. The code is generic enough to profile any application defined as a standard state machine on AWS Step Functions.

3. **DAG_Modeler:** This folder contains the code used to build the E2E performance model of the DAG. This module also contains the `VM_Size_Optimizer` to select the best VM size for each function in the DAG.

4. **Bundling_Manager:** This folder contains the code of ORION's Bundling optimization. This component of ORION profiles the DAG with varying bundle sizes and shows the P50 Latency, P95 Latency, and $ cost for each bundle size.

5. **Prewarming_Optimizer:** This folder contains the code to select the best pre-warming delays for each stage in the DAG.

6. **Comparison_to_Baselines:** This folder contains the code that compares ORION to two baselines: Best memory and CherryPick. The script produces the tail latency and cost (in $) for ORION as well as the two baselines.

7. **Policy_simulator:** This folder contains the code for our pre-warming policy simulator. This component compares different pre-warming policies without being constrained by what is possible in commercial public cloud.

Hosting

ORION is open sourced and we release its code, the workload characterization data, and the evaluation applications. All these components can be obtained at: https://github.com/icanforce/Orion-OSDI22

Requirements

The artifact uses AWS Lambda to host serverless functions, and AWS Step Functions to orchestrate the functions and organize them in a DAG. Some functions have large dependencies and hence are deployed as images on AWS ECR (Amazon Elastic Container Registry). Accordingly, users need to install the following dependencies:

1. **Amazon AWS CLI:** Can be obtained at: https://aws.amazon.com/cli/

2. **Docker:** Can be obtained at: https://www.docker.com/

Environment Setup

1. First, deploy one of the evaluation applications from `Benchmarks-AWS-Lambda` directory in AWS StepFunctions.

2. Then, use the `DAG_Profiler` to profile and generate the latency distributions for each function in the DAG.

3. Use `DAG_Modeler` to build the E2E performance model of the DAG, this module also contains the `VM_Size_Optimizer` to select the best VM size for each function in the DAG.

4. Use `Bundling_Manager` to select the best bundle size.

5. Use `Prewarming_Optimizer` to select the best pre-warming delays for the stages in the DAG.

6. Use `Comparison_to_Baselines` to compare Orion with CherryPick and Best Memory baselines.

Occualizer: Optimistic Concurrent Search Trees From Sequential Code

Tomer Shanny
Tel Aviv University

Adam Morrison
Tel Aviv University

Abstract

This paper presents Occualizer, a mechanical source code transformation for adding scalable optimistic synchronization to a sequential search tree implementation. Occualizer injects synchronization only to the update steps of tree operations, leaving traversal steps to execute unsynchronized, thereby maximizing parallelism.

We use Occualizer to create concurrent versions of a sequential B+tree, trie, and red-black tree. Evaluation on a 28-core machine shows that Occualizer's trees significantly outperform prior mechanically-crafted trees on non-read-only workloads and are comparable (within 4%) on read-only workloads. Overall, Occualizer shrinks the performance gap between mechanically- and hand-crafted trees by up to 13×. When using Occualizer's B+tree as the index in the STO main-memory database, the system's throughput degrades by less than 30% compared to the default Masstree index, and it scales better.

1 Introduction

In-memory tree data structures, or *search trees*, lie at the foundation of many systems, from databases [30, 58–60, 80] through operating systems [19–21] to storage engines [46, 68, 71]. Performance in such multicore systems depends not only on the sequential (single-threaded) speed of searching the tree, but also—often, mostly—on the scalability of the tree's *synchronization protocol*, which ensures correctness of concurrent tree operations [29, 47].

Scalable synchronization protocols typically apply *optimistic concurrency control* (OCC). In an optimistic protocol, traversals of tree paths are read-only and do not perform synchronization such as acquiring locks or executing atomic read-modify-write (RMW) instructions [11, 16, 29, 71]. Synchronization occurs only if and when an operation starts updating the tree. The optimistic approach thus limits serialization of tree operations (due to locking and/or cache coherence contention) mostly to the step that physically mutates the tree, allowing other steps to execute completely in parallel. The result is scalable performance that improves as the amount of hardware parallelism grows (unless the workload is contended at the semantic level, e.g., operations updating the same key).

Deploying an optimistic concurrent search tree in a system can be a hard problem, however. Systems often cannot deploy "off the shelf" trees, as their target use cases and workloads call for new, customized data structures [6, 17, 65, 67, 71, 80]. But designing a scalable synchronization protocol for a custom data structure—particularly an optimistic protocol—is notoriously challenging, because it involves *concurrent reasoning* to verify the algorithm's correctness under any possible thread interleaving allowed by the protocol [55, 64]. This effort also needs to be repeated whenever the data structure's algorithm changes, e.g., due to new optimizations or features.

To solve the problem of manually adding synchronization to a data structure, concurrency research has proposed *automatic transformations* such as universal constructions [2, 3, 18, 26, 35, 40, 51, 52] and transactional memory [54, 77]. These transformations receive a sequential data structure implementation (code) and produce a correctly synchronized version.

When applied to search trees, however, the automatic transformations do not produce efficient, scalable data structures. Some transformations inject pessimistic synchronization, which fully serializes all operations [18, 40, 51, 52] or all non-read-only operations [5, 26, 35]. Transactional memory-style transformations [2, 3, 31, 35, 41, 77] use optimistic synchronization, but block or restart operations whose path crosses nodes modified by a concurrent update operation, which degrades scalability. Overall, current automatic transformations produce trees whose throughput flatlines beyond 12 cores if even 3% of the workload's operations are not lookups (§ 7), as typically happens in dynamic workloads [4, 20, 58, 71].

Solution: Occualizer. This paper proposes *Occualizer*, a mechanical transformation for augmenting common sequential search tree implementations with scalable optimistic synchronization,[1] producing linearizable [56] concurrent trees. Occualizer's transformation requires the input tree to satisfy certain natural prerequisites, which most algorithms we are aware of meet, and our current prototype requires some manual effort to transform the input code. Occualizer injects synchronization only to the update steps of an operation (if any), leaving traversal steps to execute unsynchronized, unchanged from their baseline sequential code. Our key idea is to design Occualizer's injected synchronization so that it satisfies the "forepassed" condition of Feldman et al. [44]—which they prove implies the correctness of unsynchronized traversals in the presence of concurrent updates. We thus *design synchronization to satisfy a proof* instead of endeavoring to find a proof for our synchronization.

[1] Occualizer: one that adds OCC (optimistic concurrency control).

Informally, the "forepassed" condition requires that if a memory write w in the concurrent tree changes the search path for a key k, then any node v removed from the path must become immutable [44].[2] To obtain this property, Occualizer uses *localized copy-on-write* (LCOW), wherein all of an operation's writes are performed by atomically replacing the written-to nodes with new, updated copies, and making the old copies immutable. Crucially, LCOW does not require copying the entire path from the root to the updated nodes and thereby avoids synchronization bottlenecks at the top of the tree—a fundamental difference from prior COW techniques [5, 19].

We use Occualizer to produce concurrent versions of the sequential `tlx` B+tree [8], a radix tree (trie), and a red-black tree, and evaluate them on a dual-socket 28-core machine. Compared to prior transformations, Occualizer's search trees are far faster and more scalable in dynamic (non-read-only) workloads, outperforming trees using GCC's transactional memory by up to 17× and the CX universal construction [26] by orders of magnitude. On read-only workloads, Occualizer's trees are comparable to prior constructions' (within 4%). Due to instrumentation overheads, however, Occualizer's trees do not match the performance of hand-crafted concurrent algorithms. For instance, when used as an index in the STOv2 main-memory database system [58], Occualizer's B+tree has better scalability but is 25%–30% slower than the default index, Masstree [71], a hand-crafted concurrent trie/B+tree hybrid.

Overall, Occualizer significantly changes the cost/benefit analysis of hand-crafting a concurrent search tree. By shrinking the performance gap between mechanically- and hand-crafted trees by up to 13×, Occualizer makes mechanically-crafted trees applicable in many more contexts and performance targets, freeing up time and costs that would otherwise be spent on designing, implementing, and testing a hand-crafted implementation.

Contributions. We make the following contributions:

- **Transformation.** We describe Occualizer, a mechanical transformation for augmenting common sequential search tree implementations with optimistic synchronization.
- **Implementation.** We implement Occualizer and use it to produce concurrent versions of the sequential `tlx` B+tree, a radix tree (trie), and a red-black tree. Occualizer's code is available at `https://github.com/tomershanny/Occualizer`.
- **Evaluation.** We show that Occualizer's trees outperform trees using GCC's transactional memory by up to 17× and the CX universal construction by orders of magnitude, but are slower than hand-crafted concurrent trees.

[2] Intuitively, this condition guarantees that any operation whose search for k is currently located at v will either rejoin the new path or will end at an immutable node from which it cannot "damage" the tree.

2 Background, motivation, and related work

Designing efficient fine-grained synchronization for data structures is notoriously hard, because verifying synchronization correctness requires reasoning about every possible thread interleaving allowed [55, 64], while scalability requires the protocol to allow more possible interleavings. Optimistic search tree design exemplifies this challenge. On one hand, to maximize scalability, the synchronization protocol should not block or restart a traversal that encounters concurrent updates of its search path [11]. On the other hand, a traversal encountering such updates can observe *inconsistent* tree states, which cannot occur in a sequential execution but must be reasoned about to verify the protocol's correctness [43, 44, 61, 62, 66, 73, 81, 82].

The difficulty of designing a highly-scalable and correct optimistic tree lead some systems to deploy search trees with relaxed correctness guarantees. Linux's red-black tree, for instance, guarantees only that searches do not crash in the face of concurrent updates—but not search correctness [69]. Researchers have identified principles for designing optimistic synchronization protocols [11] as well as compiler support to simplify their implementation [84], but such research does not address the fundamental verification difficulty of a scalable, human-designed synchronization protocol.

Our motivation is therefore to automate the task of adding optimistic synchronization to a custom-designed sequential search tree. Concurrency research has proposed approaches for automatically transforming a sequential data structure into a concurrent one: universal constructions (§ 2.1) and transactional memory (§ 2.2). But these approaches do not produce scalable concurrent data structures when applied to search trees, as we discuss next.

2.1 Universal constructions

A *universal construction* (UC) [51] takes a sequential implementation of a data structure and outputs a linearizable concurrent version of it, without modifying the sequential code—i.e., by "wrapping" it in synchronization in some fashion. UCs can apply *nonblocking* or *blocking* synchronization.

Nonblocking universal constructions create concurrent data structures with nonblocking progress properties: either *wait-free*, which means every operation can complete in a finite number of its own steps, or *lock-free*, which means that some operation always completes after a finite number of execution steps [51]. Achieving these progress guarantees typically requires operations to coordinate and *help* each other make progress, which adds overhead [55].

Blocking universal constructions are based on the *delegation* technique [10, 15, 39, 50, 70, 74], which delegates the execution of the data structure operations threads to one thread. This "server" thread executes operations on behalf of the other, "client" threads. Delegation schemes differ in the types of operations delegated (e.g., all operations [39, 50, 70], up-

date operations [15], or read-only operations [10]) and/or their inter-thread communication techniques [14, 75].

The flip side of UCs' treatment of the input code as a black box is that the synchronization they add is coarse-grained, pessimistic, and slow. Early nonblocking UCs [18, 38, 40, 51, 52] work by having each operation execute on a local copy of the entire data structure which it then tries to install as the new version. This approach fully serializes all operations and is prohibitively slow on real-world-sized data structures. Early delegation UCs [39, 50, 70] also fully serialize all operations, as they delegate every operation to the "server" thread. Modern nonblocking UCs [5, 26, 35] improve on the full serialization aspect by optimizing read-only operations, allowing them to execute in parallel, but non-read-only operations are still serialized. Likewise, modern delegation UCs allow certain types of operations to execute in parallel (updates [10] or read-only operations [15]), but all other operations remain serialized by delegation.

COW UC. A nonblocking UC technique that reduces copying overhead (and is closely related to Occualizer) is copy-on-write (COW), used in the transactional system of Ben-David et al. [5]. The core idea is for a writing operation to create its updated version without copying the entire data structure, by having it share as much as possible with the previous version. For trees, this technique updates a node by creating an updated version of the node and the path that leads to it, and atomically swapping the updated root with the new one (Figure 1). The COW approach allows read-only operations to proceed without synchronization, since the version they observe is immutable. Writing operations, however, remain serialized.

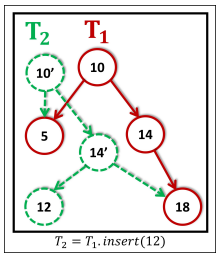


Figure 1: COW: Adding 12 as a child of 14 in T_1 yields T_2.

2.2 Transactional memory

Transactional memory (TM) [54, 77] executes sequential code segments as isolated atomic transactions. With hardware TM (HTM), serializability of the transactions is enforced by the hardware [54]. HTM can thus be viewed as a UC. Real-world HTM extensions, however, have several limitations [33, 33, 34] and are currently disabled on many processors due to hardware errata [63]. We therefore focus on software TM (STM). STMs differ from UCs in that they require *code instrumentation*, so that the STM runtime can mediate reads/write to memory and (in some cases) memory allocation/deallocation.

Modern STMs have converged on designs using optimistic-style lock-based synchronization [36, 48]. In these designs, the STM algorithm performs transactional reads without writing to memory (e.g., to acquire a lock); writes either acquire locks ("eager" locking) or are buffered in a write set ("lazy" locking). When the transaction ends, the STM checks whether there is a point in time in which all of the transaction's reads and writes can appear to take place atomically. If so, the transaction *commits* and its writes are made visible to other transactions (e.g., locks are released). Otherwise, the transaction *aborts* and must restart.

Unfortunately, since the STM does not understand the semantics of the underlying code, its validation conservatively depends on every value read by the transaction [31, 41]. Therefore, if any memory location read by a transaction is written to before the transaction commits, the transaction will abort. This effect severely limits scalability of STM-based trees, because any concurrent write to an operation's search path causes the operation to abort—even if the operation would have reached the same location in the tree had it executed on the new path (a fact the STM cannot know). In our experiments, TM performance can flatline at low core counts even if as few as 3% of the tree operations are updates (§ 7).

TM research has proposed several approaches to address the above problem. First, an STM can determine the serial order of transactions (*conflict detection*) more intelligently [76, 85]. But this typically requires transactional reads to write to memory, which can lead to undesirable serialization of readers. Second, transactions can be built over higher level objects instead of low-level memory reads/writes [49, 53, 57]. But this requires designing the underlying thread-safe objects, which was our original problem. Finally, transactional semantics can be relaxed [42] to avoid aborting a transaction in cases such as search path changes. But then one has to prove the resulting relaxed transactions correct, which requires the concurrent reasoning about thread interleaving that we wish to avoid.

2.3 Summary and goals

In summary, there is still no mechanic way to transform the source code of a sequential tree implementation into an optimistic, fast, and scalable concurrent tree—without needing to perform concurrent reasoning to verify the correctness of the produced concurrent code. This is our goal.

Occualizer sits in the middle between UCs and TM. Compared to universal constructions, Occualizer takes a pragmatic approach. Instead of accepting arbitrary sequential code as input, Occualizer requires the input to have certain natural prerequisites, and also transforms/instruments the sequential code. Compared to transactional memory, Occualizer is specialized to search trees, which enables us to design an optimistic synchronization protocol that does not restart operations whose search path is modified by concurrent operations.

3 Occualizer Overview

Occualizer receives source code of a sequential (single-threaded) search tree and transforms it into an optimistic concurrent implementation by adding calls to Occualizer's synchronization library into the input source code. This section gives an overview of the Occualizer transformation.

We first define the family of sequential tree implementations to which Occualizer is applicable (§ 3.1). (We discuss verifying that a sequential tree meets Occualizer's requirements in § 3.5.) We then give an overview of Occualizer's source code changes (§ 3.2) and the run-time synchronization protocol they inject, called localized copy-on-write (§ 3.3). The details are described in §§ 4–5. Finally, we outline the correctness proof of the produced concurrent tree (§ 3.4), which appears in § 6.

3.1 Scope

We consider correct sequential implementations of a dictionary datatype, namely, that provide lookup, insert, and delete operations on key-value pairs. The implementation may also (optionally) support ordered iteration over the stored keys, provided via key predecessor/successor operations. We assume a programming language with manual memory management. (Our prototype targets C++.)

Occualizer requires the sequential input algorithm to meet certain prerequisites (PRs), detailed below. At a high level, the prerequisites are that (1) tree operations are composed of a read-only traversal followed by reads and writes which are determined only by what the operation observes after the traversal; (2) each step in the traversal depends only on the target key and the current node; and (3) any operation that moves a node v off some search path(s) must also access v.

The user is responsible for verifying that the input meets Occualizer's prerequisites, and the concurrent tree produced by Occualizer is not guaranteed to be correct if they are not met. The human effort required for this verification (and the possibility of errors there) are limitations of Occualizer compared to general universal constructions that accept arbitrary code. While our experience has been that the prerequisites are met by many algorithms and that verifying them requires reasonable effort (see § 3.5), our vision is to develop automated verification of the prerequisites to fully automate Occualizer.

Prerequisites. We define the prerequisites in terms of an algorithm maintaining a directed graph G of nodes, whose edges represent pointers between nodes.

PR1 Maintain a rooted tree: At the end of any sequence of operations, the graph G is a rooted tree.

Crucially, PR1 does not care about intermediate states that occur while a tree operation executes, only about the graph's structure upon its completion. PR1 is conservative, as Occualizer can support structures with auxiliary edges linking nodes to their successor/predecessor, which create multiple paths from the root to nodes and so are not formally trees. We defer these details to § 4.

PR2 Read-only traversals: Every operation $op(k)$ consists of a read-only traversal $traverse(k)$ that searches for k followed by read/write steps.

PR2 is not met by self-balancing trees that perform balancing during traversals, such as splay trees [78]. But PR2 is met by self-balancing trees such as red-black, AVL, or B-trees, which perform self-balancing after updating the tree (post-traversal).

PR3 Traversals are single-step: The next node visited by $traverse(k)$ depends only on k and on the current node.

PR3 is met by trees with comparison-based traversals, such as B+trees [22], Bw-Trees [67, 83], red-black and AVL trees, etc., where how $traverse(k)$ proceeds depends only on how k compares to the key(s) of the current visited node. PR3 can also be met by tries, provided that nodes encode the key offset they represent; otherwise, the next node visited becomes dependent on all the nodes visited so far, which violates PR3.

Our next prerequisite states that the reads and writes an operation performs after its traversal are not a function of observations made during the traversal:

PR4 Post-traversal actions depend only on subgraph accessed post-traversal: Consider an operation $op(k)$ that executes on tree T, whose $traverse(k)$ finishes at node v. Let $RW^{op(k)}$ be the set of nodes read/written to by $op(k)$ after finishing its traversal. Let $T_{RW^{op(k)}} \subseteq T$ be the smallest subgraph of T containing $RW^{op(k)}$. Then for any sequence of operations that execute on T resulting in tree T', if $T_{RW^{op(k)}} \subseteq T'$ and $traverse(k)$ executed on T' finishes at the same node v as in T, it holds that running $op(k)$ over T' results in exactly the same reads and writes as in op's execution over T.

PR4 does not preclude an algorithm from reading or writing parts of its search path after completing the traversal—as in, e.g., rebalancing of red-black, AVL, and B-trees—because any node on the path that an operation $op(k)$ reads or writes post-traversal becomes part of $RW^{op(k)}$. PR4 is thus met by classic tree algorithms that perform post-traversal rebalancing.

PR5 Moving off a search path implies a post-traversal access: For any operation op and any key k, consider the paths P and P' that would be taken by $traverse(k)$ before and after op executes. Then if $v \in P$ but $v \notin P'$, op must read, write, or destroy v after its traversal.

For an implementation that does not expose nodes to its client, PR5 is met by every tree algorithm we are aware of. In these algorithms, a node moves off a search path due to either (1) a structural modification that changes the node's position in the tree, in which case the node is written and/or read; or (2) being removed from the tree, in which case the implementation destroys and frees the node, as it has no other references.

Occualizer for managed languages. Occualizer's design and its prerequisites are programming language agnostic. Most tree implementations, however, fail to meet PR5 when implemented in a managed language. The reason is that

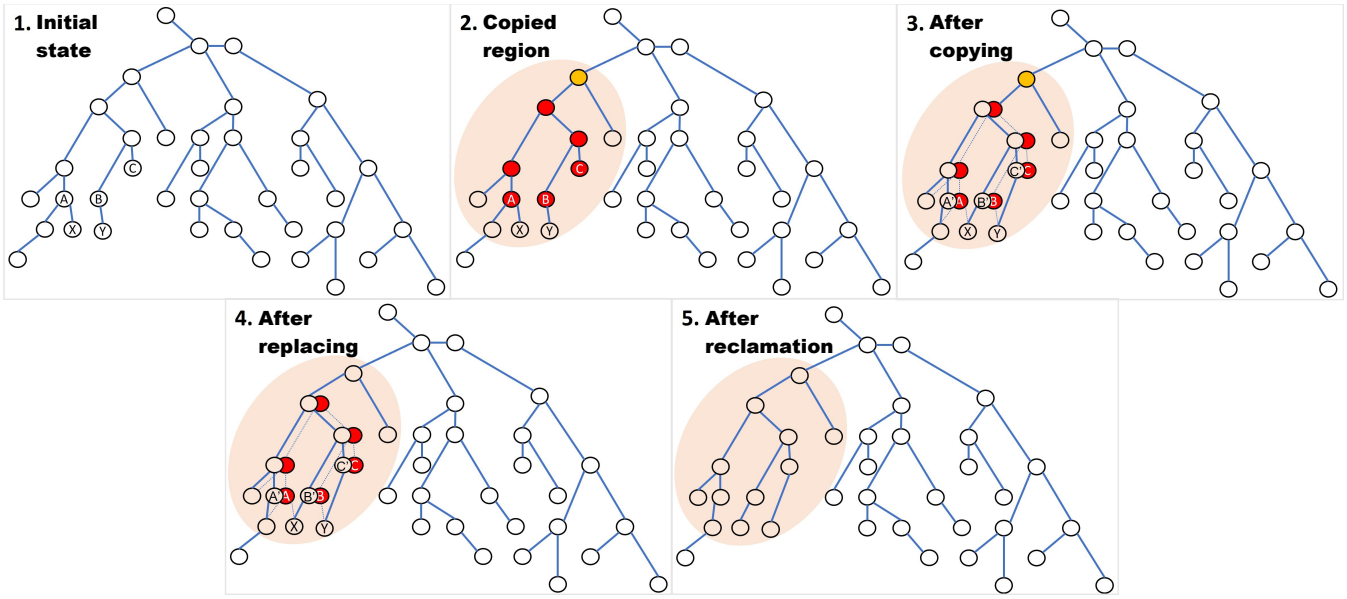


Figure 2: Illustration of LCOW on an operation that makes B and C the parents of X and Y, respectively.

node destruction in managed languages is performed asynchronously by the garbage collector and not explicitly by the program, so a managed tree implementation meets PR5 only if it writes to a node when removing it from the tree—which most algorithms do not do. This problem can be fixed (currently, manually) by adding no-op writes to removed nodes.

3.2 Code transformations

Occualizer transforms the sequential input code by adding calls to Occualizer's synchronization library. These calls are similar to object-level transactional memory instrumentation. They include calls to demarcate each operation's start and completion, and to access (read/write) node fields only through the library's interface. Field accesses are captured straightforwardly by requiring the sequential input code to access fields using only getter/setter methods, which the transformation then replaces. The transformation also adds locks and metadata fields to the node structure.

The code transformation is mechanic and our design is for it to be done automatically, with minimal user involvement. In our current prototype, however, we implement only the synchronization library and perform the code transformations of the evaluated trees manually (following the mechanical recipe given in § 4). Implementing the automatic code transformation is an ongoing effort.

3.3 LCOW synchronization library

Occualizer's code transformation leaves the logic of traversals unchanged. In particular, traversals do not block or retry mid-operation. The synchronization added to writing operations guarantees the correctness of both traversals and writing operations. To this end, the library uses a technique we call *localized copy-on-write* (LCOW). LCOW exposes all of an operation's writes atomically, using one atomic write. Unlike other COW techniques [5, 19], this write *does not* typically target the tree's root and thereby avoids creating a synchronization bottleneck.

LCOW works as follows. Once an operation *op* finishes its traversal, the library uses a combination of locking and validation checks to maintain an invariant that *op*'s further observations of the tree are consistent with some sequential execution. This invariant is needed to guarantee that *op*'s code behaves correctly. In particular, whenever *op* first writes to some node v, the library locks v and creates a copy of v, v'. (If a lock acquisition fails, *op* is restarted, releasing any locks it holds and freeing node copies it had made.) Subsequently, all of *op*'s accesses to v are redirected to v', ensuring *op* "sees its own writes." When *op* completes, the library identifies a minimal subgraph containing all written nodes, called the *copied region*. This subgraph is itself a tree rooted at some node n, but it may not be n's subtree (i.e., it may not include all of n's descendants). Next, the library locks and creates a copy of the copied region, updated to contain the nodes written to by *op*. Finally, the library exposes *op*'s writes atomically by linking u', the root of the copied region, instead of its original version u with one atomic write. Crucially, the old versions of the nodes remain locked, making them immutable.

The library reclaims the memory of the old copied region only once it is guaranteed that no concurrent operation may be accessing the old region, using a read-copy update (RCU) epoch-based memory reclamation scheme [45, 72].

Figure 2 illustrates LCOW on some abstract operation *op*. ① shows the initial tree state. Assume that executing *op*'s sequential code from start to finish in this state would make

B and C the parents of X and Y, respectively. ② shows the copied region: LCOW locks the orange and red nodes, thereby blocking concurrent modifications to every node in the pink circle. ③ LCOW copies the red nodes, creating a new version of the copied region in which *op*'s writes are made to the nodes A', B', and C'. Finally, ④ shows the memory state after atomically replacing the copied region with its new version, and ⑤ shows the memory state after the original copied region is reclaimed.

The upshot is that Occualizer guarantees that (1) executing operations run correctly, as they would in a sequential execution, and (2) the state of the tree in memory is always a state that can be produced by a sequential execution. Crucially, however, this is all achieved while still allowing the *traversal* part of an operation to observe an inconsistent state. E.g., a traversal can start in tree T_1 and then cross into tree T_2, walking a path that never actually existed in memory.

3.4 Linearizability argument

Trees produced by Occualizer are linearizable [56], i.e., operations appear to execute atomically. The main challenge of proving linearizability is that because traversals are unsynchronized and can observe inconsistent tree states, it is not clear that a traversal ultimately reaches the correct node.

The key idea of Occualizer is to design its synchronization to satisfy the precondition of an existing proof (from the concurrency literature) that an unsynchronized traversal is correct. Occualizer's trees satisfy the "forepassed" condition, which Feldman et al. [44] prove implies that if an unsynchronized traversal searching for key k reaches node v, then at some point during its execution, v was on the search path for k. (That is, the state of the tree was such that had the traversal executed from start to finish then, it would have reached v.)

The above immediately proves the linearizability of read-only lookups that consist only of traversals. To prove linearizability of writing operations, we show that when a writing operation atomically performs its writes using LCOW, then the state of the tree is such that had the operation's sequential code executed atomically now, it would have behaved exactly the same. In other words, the state of the tree in memory remains consistent with some sequential execution of the original sequential code. We show this by first proving an invariant that an operation locking node v implies that v is on the relevant search path at lock acquisition time ("now"). The proof then follows from PR4, since the copied region locked by an operation contains the subgraph it accessed post-traversal.

3.5 Discussion: Prerequisite verification

For our evaluation (§ 7), we use Occualizer on sequential implementations of classic tree algorithms, such as the B+tree. We draw on this experience to discuss the effort and reasoning needed to manually verify that an implementation meets Occualizer's prerequisites. In a nutshell, we find that the prerequisites are met by many algorithms (e.g., red-black and B-trees) and tree design techniques. We also find that checking the prerequisites requires reasonable effort, given basic understanding of the input tree's algorithmic properties. In particular, there is no need for concurrent reasoning, as the prerequisites are properties of sequential code.

Verifying PR1–PR3 involves straightforward code inspection. In particular, verifying that every tree operation begins with a read-only traversal (PR2) is easy for implementations with an explicit traversal method and for recursive implementations, where one only needs to check the recursive function.

Verifying PR4–PR5 requires reasoning about the principles driving the sequential input algorithm. To verify that post-traversal actions depend only on the subgraph accessed post-traversal (PR4), we need to check that the nodes and fields an operation chooses to access and the values it writes depend only on what it reads after its traversal. PR4 would be violated, for example, by an operation writing a node's depth (distance from the root) that was computed while searching for the node. On the other hand, PR4 is satisfied by an operation maintaining the height of a node (or the balance factor in an AVL tree) using a bottom-up computation after the traversal. PR4 holds trivially if traversals are performed as a subroutine call that returns only the target node, thereby making the search path opaque to the operation.

To verify that if a node stops being on the search path for some key k, then the node must be accessed or destroyed (PR5), we need to verify that a node removal destroys it, and to reason about how tree structure modifications affect the behavior of searches. We find that common tree algorithmic techniques meet PR5. For instance, consider a binary tree rotation [25] moving node y above its parent y (Figure 3). The only node that moves off a search path as a result of the rotation is y (which moves off the paths leading to subtree A) and y is indeed written by the rotation (its left child changes).[3] As another example, in a binary tree that deletes an internal node by replacing it with its successor [25] (the leftmost node of its right subtree), the nodes on the path to the successor move off the search path to the successor. These nodes are read by the removing operation as it searches for the successor, so PR5 is met.

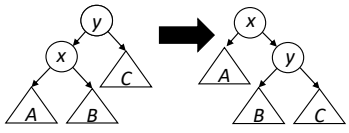


Figure 3: Rotation moving x above y.

4 Design

This section describes Occualizer's design. We first describe Occualizer's synchronization library interface and the mechanical rules for calling its methods from a sequential tree implementation (§ 4.1). We next describe how the library

[3] Crucially, PR5 depends only on the effect that a complete rotation has on future searches—not on the exact order of writes performing the rotation in the sequential code.

implements the LCOW synchronization protocol (§ 4.2) and then extend the design to support algorithms with auxiliary edges between nodes, which are typically used to optimize iteration over nodes (§ 4.3).

4.1 Library interface & code transformations

Interface. Occualizer augments the input with calls to its library. Table 1 describe the library's transactional memory-like interface, which consists of "macro" and "micro" methods.

The "macro" methods demarcate the points in which the operation starts/finishes and where its traversal ends. In particular, `occ_start` checkpoints the calling thread's register state (e.g., with `setjmp()`) and restores it if the library decides to abort the operation. When an abort happens, `occ_start` returns a failure indication.

The "micro" methods read and write node fields (or the root pointer) and notify the library of allocated or destroyed nodes. For simplicity, we show the read/write methods as taking the field name as an argument. An implementation either generates specific methods for each field or has a general method that takes the field's offset and size in the node.

Transformation. Transforming a sequential tree to an optimistically-synchronized one using Occualizer requires two types of transformations. Macro transformations add macro calls to demarcate each operation with `occ_start`, `occ_traverse_done`, and `occ_finish` calls, and restart it after an abort (Listing 1). Occualizer does not require modifying the input code to separate the traversal into its own method, only to call `occ_traverse_done` when it is done. This property allows the operation's subsequent code to reuse information learned during the traversal, e.g., to climb back up the path for tree maintenance.

Micro transformations replace calls to node setter/getter methods with the appropriate `occ_set`/`occ_get` calls, and add `occ_node_born`/`occ_node_dies` calls to the node constructor/destructor.

Method	Called when (and why)
`occ_start`	Operation starts (to initialize bookkeeping data)
`occ_traverse_done`	Traversal finishes (to start consistency checks)
`occ_finish`	Operation finishes (to atomically perform operation's writes)
`occ_restart`	Restarting an aborted operation (to free resources acquired during the failed execution)
`occ_set`(n, f, v)	Writing $n.f \leftarrow v$ (to lock and copy n)
`occ_get`(n, f)	Reading $n.f$ (to read from n's copy, if it exists)
`occ_node_born`(n)	Node is allocated
`occ_node_dies`(n)	Node is destroyed

Table 1: Occualizer synchronization library interface.

```
Function transformed<op>(args):
    while True do
        if occ_start() then
            result ⟵ op(args) ; ▷ occ_traverse_done was
              added inside op's code
            if occ_finish() then
                return result
            end
        end
        occ_restart() ; ▷ Op aborted
    end
```

Listing 1: Code of macro-transformed operation *op*.

Mechanizing the transformation. The transformation can be performed automatically by a source-to-source transformer tool, which we are in the process of implementing. The transformer requires the user to supply the tree's sequential source code, the names of methods to be macro-transformed and structure(s) implementing nodes, and to manually add the `occ_traverse_done` call. The transformer performs the following steps: ① Ensure that all node fields are accessed via setter/getter methods, by replacing every direct field access with an appropriate setter/getter call and generating getter/setter methods if they do not exist in the input code. ② Perform the micro-transformations by modifying methods in the node structure. ③ Generate the macro-transformed operations.

4.2 LCOW synchronization library

Occualizer's synchronization library has two high level tasks. First, it tracks the tree as observed by the operation, so that once the operation's traversal finishes, Occualizer can guarantee that the tree is in a consistent state from the operation's perspective. Second, the library buffers the operation's writes and exposes them atomically when the operation completes. We now walk through the library's flow.

Initialization (occ_start). This method checkpoints the thread's local state, so that execution can restart if the operation subsequently aborts. It then initializes the library's thread-local bookkeeping variables (Table 2), which track edges observed by the operation, nodes allocated, destroyed, locked, and copied, and other flags, such as whether the operation is in the midst of its traversal. We treat these variables as abstract datatypes for now (in particular, without considering implementation efficiency); § 5 describes our implementation.

Node reads (occ_get). We first focus on the case of reading a node pointer (child), i.e., reading an edge. As long as an operation is traversing the tree, its reads are handled with minimal overhead. The library only stores traversed edges in the `edgeSet`, to verify their consistency in case the operation rereads them in its post-traversal steps. Once the traversal

Name	Type	Content
`edgeSet`	Set of edges	Edges observed during operation
`lockedSet`	Set of nodes	Nodes locked by operation
`copySet`	Set of node pairs	Copied nodes and their copies
`bornSet`	Set of nodes	Nodes allocated by operation
`destroyedSet`	Set of nodes	Nodes destroyed by operation
`traversing`	Boolean	Initially *True*; *False* after `occ_traverse_done` called

Table 2: Thread-local bookkeeping structures. The term "node" refers to a pointer to the relevant object in memory.

completes, the library switches to a mode that guarantees consistency of the observed tree. This is accomplished by locking any node accessed post-traversal, while verifying that the locked node belongs to the most updated tree in memory. If this verification fails, the operation is aborted (releasing any locks it acquired and nodes it allocated) and restarted.

Listing 2 shows the pseudo code of `occ_get`. Suppose operation *op* is trying to read the c-th child of node n. (For generality, we assume child pointers are stored in a `children` vector in the node.) If *op* previously created a copy of n, the read is satisfied from the copy n' without further checks. Otherwise, n's c-th child, u, is read from n. If *op* is traversing, the method *trackEdge* only remembers the edge (n,c,u) in `edgeSet`. Otherwise, *trackEdge* locks n by calling *lockNode*.

The *lockNode* method locks n while checking its consistency with *op*'s observations of the tree. If n is not present in *op*'s `lockedSet`, *op* tries to acquire n's lock. If n is locked, *op* aborts, to avoid deadlocks. Then *op* checks that any edge into or out of n that *op* previously observed still exists. If so, n is added to *op*'s set of locked nodes. Otherwise, *op* aborts.

Reads of non-pointer fields are handled identically (locking the node, etc.) except that the read values are not tracked.

Node writes (occ_set). On any write to a node n, the library locks n and creates a copy of it, n'. If the operation completes successfully, n' will take the place of n in the tree and n will remain locked and hence immutable until its memory is reclaimed. Listing 3 shows the pseudo code of `occ_set`, again focusing on the case of writing a child pointer. If *op* has made a copy, v', of the pointed-to node v, the value to be written is changed to v'. Next, *op* checks if the written node n is part of the tree, i.e., it was not allocated by *op* itself and is not a copy. If so, *op* creates a copy of n by calling *occ_create_copy*. Finally, the write is performed.

To copy n, the *occ_create_copy* methods locks n using the *lockNode* method described above. It then copies the (now locked) n into a new node, n', records that n has a copy n' in *op*'s `copySet`, and finally adjusts the links between the existing copied nodes to reflect the new copy. For every $(x,x') \in$ `copySet`, the *fixLinks* method changes any edge

```
Function occ_get_child(n, c):
    if (n,n') ∈ copySet then
        return n'.children[c]
    else
        u ← n.children[c]
        trackEdge(n, c, u)
        return u
    end

Function lockNode(n):
    if n ∈ lockedSet then
        return
    if tryLock(n.lock) fails then
        abort operation
    ▷ Validate
    foreach (x,c,y) ∈ edgeSet, s.t. x = n or y = n do
        if x.children[c] ≠ y then
            abort operation
    end
    lockedSet.add(n)
```

Listing 2: Code of reading a node child..

```
Function occ_set_child(n, c, v):
    if (v,v') ∈ copySet then
        v ← v'
    end
    if n ∉ bornSet and (n,_) ∉ copySet then
        n ← occ_create_copy(n)
    end
    n.children[c] ← v

Function occ_create_copy(n):
    lockNode(n)
    n' ← copy of n
    copySet.add(n,n')
    fixLinks(n,n',copySet,bornSet)
    return copy
```

Listing 3: Code of writing a node child.

pointing from x' to n to point to n' instead, and changes any edge from n' pointing to x to point to x' instead. It similarly fixes any edge pointing to n from nodes allocated by *op*.

Writes of non-pointer fields are handled identically, except that value written is not "translated" as it is not a node pointer.

Traversal completion (occ_traverse_done). On traversal completion, the library switches to its post-traversal mode, in which any accessed nodes is locked. In addition, the last node read by the traversal is locked using the *lockNode* method. This locking is needed to guarantee that concurrent tree modifications do not "invalidate" the node's traversal (see § 6).

Node allocation/deallocation. On node allocation, the new node n is added to `bornSet`. On node destruction, the destroyed node n is locked (using *lockNode*) and added to `destroyedSet`. This is done so that n can be left immutable when the operation completes.

Operation commit (occ_finish). This method *commits* an operation by atomically applying its writes to the tree. Consider first the simple case in which the operation *op* finishes its traversal at node n and subsequently accesses nodes only down the tree. In this case, the paths to nodes written by *op* form a tree T_n rooted at n, and all nodes in T_n are locked by *op*. Occualizer can thus atomically apply *op*'s writes by creating a copy of T_n, T_n', which contains the updated versions of the nodes *op* wrote to and then swinging the pointer to n from its parent, p, to point to n', the root of T_n'. (Figure 2, with p being the orange node.) We thus call T_n the *copied region*.

To ensure atomicity, *op* must verify that p is still in the tree and that the edge (p,c,n) that it swings has not changed since *op* originally crossed p during the traversal. If n is a non-root node, this is done by locking p and validating that $p.children[c] = n$ before overwriting it. If n is the root, this is done by locking a global root lock and validating the root's value (which also gets saved in `edgeSet`). Although we describe this parent validation step last, it chronologically occurs before copying any yet uncopied nodes from T_n, to avoid wasting CPU cycles in case *op* is doomed to abort.

Finally, nodes locked by *op* that are not in T_n are released. The nodes of T_n are *retired*, which means that their memory is freed/reclaimed once no concurrent operation can observe them. (Occualizer relies on an RCU-like epoch-based safe memory reclamation (SMR) management library [45, 72] to provide this functionality.) Until reclamation, these nodes remain locked and thus immutable.

In the general case, *op* may have proceeded up the tree, by accessing nodes it observed while traversing. The above discussion still holds, except that instead of taking n as the copied region's root, *occ_finish* needs to find the lowest common ancestor (LCA) of all written nodes and lock every path from that LCA to each written node. This is straightforward to do, because the LCA and all relevant edges have been read during *op*'s run (possibly only in the traversal step).

4.3 Optimizing range scans

An important feature of search trees is that they support *range scans*, the ability to iterate over the stored keys in order by using successor/predecessor calls. Specifically, we assume the tree implements a C++ standard library (STL)-like `iterator` object. The iterator maintains a key k', which is initially the predecessor or successor of its constructor argument. The iterator provides next/prev calls, each of which updates k' to its successor/predecessor, respectively. As with other concurrent search trees [9, 71], our goal is for the individual next/prev calls to be atomic (linearizable)—not for an entire range iteration performed by a sequence of such calls to be atomic with respect to insertions/deletions.

In sequential trees, a common method of implementing an iterator is to add *auxiliary* next/prev pointers to node fields, so that advancing an iterator does not require walking paths in the tree. Occualizer supports trees with such auxiliary edges, provided that they are symmetric (i.e., v points to u via an auxiliary link if and only if u points to v), as is the case of next/prev pointers. In addition, the user is required to specify the field names of auxiliary links in the node structure. Occualizer then leaves iterator movement over auxiliary edges as a read-only operation.

Extended commit protocol. We extend Occualizer's commit protocol to support auxiliary edges as follows. When an operation *op* is ready to commit, after having locked and validated p, the parent of r, the copied region's root, *op* checks each auxiliary edge (v,u) pointing to the copied region (i.e., such that u is in the copied region and v is not) and attempts to lock v (aborting if it fails).[4] Once all these "border" nodes are locked, *op* updates p to point from r to its new version r' with one atomic write, and then iterates over each locked border node v, updating its relevant auxiliary edges to point to the new version of the neighbor u (from (v,u) to (v,u')), and releasing v's lock afterwards. This protocol may seem heavyweight, but in practice copied regions tend to be small, so the extra cost of handling auxiliary edges is not substantial.

Unfortunately, the extended commit protocol breaks Occualizer's LCOW technique of replacing a copied region with one atomic memory write. The problem is that a sequential tree with auxiliary edges does not meet our PR1, because the auxiliary edges create more than one path to a node, and the commit protocol needs to update these paths when replacing a copied region. For example, in an external binary tree whose leaves are connected with next/prev links, Occualizer needs three writes to replace a copied region—to the parent of the region's root and to the predecessor and successor nodes of the region's leftmost and rightmost leaves, respectively.

Because Occualizer cannot *physically* atomically update all edges crossing the border between a copied region and the rest of the tree, our solution is to make the update *logically* atomic, as detailed below.

Logically atomic updates. To make iterators observe updates atomically, we ensure that an iterator only moves across auxiliary edges that exist in the latest version of the tree—i.e., edges that are not part of, or cross into, a copied region which is being replaced. To achieve this, Occualizer prevents iterators from moving to a locked node, relying on the fact that every node in a copied region is locked. When an iterator positioned at node v attempts to move to v's neighbor u and finds either v or u locked, the iterator instead "resynchronizes" its position using the latest version of the tree. Specifically, the iterator searches from the root for v's predecssor or successor x (as during iterator construction), according to where the iterator was trying to move. The iterator then positions itself at x and returns x's key.

This protocol guarantees that after an updating operation *op* exposes its writes (by updating some child pointer to link

[4] Our requirement that auxiliary edges are symmetric guarantees that *op* finds every node with an auxiliary edge to the copied region.

the new version of *op*'s copied region into the tree), no iterator can move into the copied region (whether the iterator is positioned inside or outside the copied region). Attempting such a move causes the iterator to "resynchronize" itself in the updated tree, which no longer contains the copied region. The protocol is conservative, however, in that while *op* holds the copied region locked but has not yet exposed its writes, an iterator moving across an auxiliary edge (v,u) to such a locked node u will "resynchronize" superfluously, ending up at u again, as it remains reachable from the root. Such a superfluous "resynchronization" does not violate the iteration's correctness; it can be thought of as reaffirming the iterator's position in the tree, with the iterator's move being linearized before *op*'s updates.

5 Implementation

In our Occualizer prototype, we perform the input code transformations manually (following the recipe of § 4.1) and implement the synchronization library in C++ using `pthread` spinlocks. This section describes the library's implementation.

Thread-local structures. We implement the various sets using thread-local dictionary (unordered map) objects, for efficient access. The `copySet` is implemented as a pair of maps, from nodes to their copies and vice versa. The `edgeSet` is implemented as a pair of maps, an *incoming* map that maps a node to its parent and child index there, and an *outgoing* map that maps a node to a list of its children indices read. The `lockedSet` is implemented as a map from locked nodes to a boolean indicating if the lock should be released on commit. The other sets are implemented as C++ STL vectors.

Efficient `copySet` searches. We further optimize `copySet` searches by adding a flag into the node structure which is set when a node is copied. The `occ_get` method uses this flag to avoid superfluous `copySet` searches. When an operation aborts, it clears this flag from all the nodes it copied.

Optimized dictionaries. Our initial implementation used C++ STL hash tables (`unordered_map`), but we observed that they impose considerable overhead. We therefore replace them with an optimized design, which initially inserts items into a small STL vector, and if the vector becomes full, stores overflowing items in an `unordered_map`. The observation underlying this optimization is that in most tree algorithms, the thread-local data structures Occualizer maintains will be small. But for correctness, we must support worst-case behavior in which these structures may contain every node in the tree. Overall, this optimization improved the throughput of an Occualizer B+tree by a factor of two.

Correctness testing. We use a couple of testing techniques to gain confidence in the correctness of the Occualizer prototype. First, we use the linearizability checking option of the SetBench [13] benchmarking harness (§ 7.1). With this option, SetBench verifies that every successful insert/delete operation during the execution is correctly reflected in the final state of the tree—so that, for example, inserted items were not lost or inserted more than once. We test different tree sizes, to test executions with varying contention levels and thread interleavings. Second, we check that tree-structural invariants of the sequential implementations we transform (e.g., the red-black property of a red-black tree) hold, both at random times during the execution and after it completes.

6 Correctness

This section sketches the proof that trees produced by Occualizer are linearizable [56], i.e., tree operations appear to execute atomically. We consider the shared-memory system running the tree. A *state* of the system consists of the memory state (contents of each address) and the local states of each thread. In each *step* of the execution, some thread accesses memory, and as a result, its internal state and/or memory change.

Our proof works as follows. We first show that traversals are correct. That is, if $traverse(k)$ stops at node v in the concurrent execution, then at some point during its run, the memory state σ was such that had $traverse(k)$ executed atomically (from start to finish) on σ, it would also reach v. We denote this property of a state σ by $\sigma : \overset{k}{\rightsquigarrow} v$. We then use traversal correctness and Occualizer's synchronization protocol to show that if an operation *op* commits in memory state σ, then had *op* run from start to finish on σ, it would have executed exactly the same. Hence, *op* appears to execute atomically at σ.

Showing traversal correctness is hard, because traversals are unsynchronized and can observe inconsistent tree states. We solve this problem by applying the theorem of Feldman et al. [44], which says that in a concurrent tree satisfying a "forepassed" condition, unsynchronized traversals are correct. The "forepassed" condition requires (1) traversals to be single-step, which we satisfy by PR3; and (2) that if the concurrent algorithm performs a write w moving the system from state σ to σ', such that $\sigma : \overset{k}{\rightsquigarrow} v$ but $\sigma' : \overset{k}{\not\rightsquigarrow} v$ for some k and v, then v is never modified later.

Requirement (2) above follows from PR5 and the fact that Occualizer leaves written/destroyed nodes immutable (locked). There is a subtle issue, however, which is that the prerequisites are met by the sequential code, so unless we know operations in the Occualizer tree behave correctly, we cannot rely on the prerequisites. But we need traversal correctness to prove this fact, creating a "chicken and egg" problem.

We address this problem using a proof technique suggested by Feldman et al. for proving "forepassed" is satisfied [44, §7]. The technique is to prove both that "forepassed" is satisfied and that the concurrent tree is correct in tandem, inductively (on steps of the execution), so that each proof can rely on the other property holding on the execution thus far.

Accordingly, we prove correctness of an Occualizer tree assuming traversal correctness. The proof is inductive: we need to show that in every state σ, if $op(k)$ runs sequentially on σ, its post-traversal memory accesses are identical to its post-traversal memory accesses so far.

In the base case, σ is when *op*'s traversal's stops at node *v*. Traversal correctness implies that for some σ' during *op*'s execution, $\sigma' :\overset{k}{\rightsquigarrow} v$. Now, Occualizer locks *v*. Based on the induction hypothesis, a lock acquisition implies that $\sigma :\overset{k}{\rightsquigarrow} v$—i.e., *v* is on the search path for *k* "now"—as otherwise, *v* would have been locked and copied between σ' and σ and *op*'s lock acquisition would have failed. Thus, if *op* runs sequentially at σ, its traversal would reach *v*. But the lock acquisition then implies that the traversal would also *stop* at *v*, since only pointer fields in *v* could have changed, but *op* verifies that any pointer it read did not change after locking *v*.

The inductive step is similar. We are in state σ with *op* successfully locking some node *u*. Inductively, we know that (1) in some σ' in the past, the state of the tree was such that *op*'s execution on it would have lead it to its current local state, and (2) any modifications made to the tree since σ' have only moved it through consistent states. Moreover, due to *op*'s locking, these changes in tree state do not change the nodes and edges *op* has observed post-traversal. It follows from PR4 that if we run *op* in state σ, it will behave exactly the same. This concludes the overall proof, when σ is the state in which *op* commits its writes.

7 Evaluation

We compare Occualizer trees to mechanically-crafted trees (§ 2) and to hand-crafted trees, with respect to scalability, throughput, and memory use. We first evaluate the trees on workloads with different amounts of writing operations (§ 7.1). We then focus on an Occualizer B+tree: we compare it as an index in the STOv2 main-memory database to the default Masstree index (§ 7.2), and analyze its overhead (§ 7.3).

Transformed trees. We use Occualizer to create concurrent versions of the following sequential trees:

- **B+tree**: An improved version of the optimized STX in-memory B+tree [7] taken from the `tlx` library [8].
- **Radix**: An implementation of a radix tree (trie) [79]. The code follows the description of Linux's radix tree [24].
- **RB**: A red-black tree [32]. The code is the sequential implementation used in Synchrobench [47].

We refer to a transformed tree implementation *T* as *occ[T]*.

Experimental platform. We use a dual-node NUMA server. Each node has a 14-core Intel Xeon Gold 6132 (Skylake) processor and 96 GB of DDR4-2666 DRAM. Hyper-Threading and Turbo-Boost are disabled. Threads are split between the nodes and memory allocation is interleaved across the nodes. Code is compiled using GCC 8.3.0 and linked with the `jemalloc` [37] multi-threaded memory allocator. Reported numbers are averages of 10 runs; all measurements are within $\pm 5\%$ of the average.

7.1 Contention benchmarks

We compare Occualizer's trees to mechanically- and hand-crafted trees on workloads with increasing amounts of writing operations.

Trees. We compare to the following trees, which unless noted otherwise are mechanically-crafted from the same sequential code used for Occualizer:

- **Global-Lock**: Created by serializing operations with a global lock.
- **GCC-TM** [1]: Created by wrapping operations in transactions using GCC's transactional memory (TM) support. The underlying TM algorithm uses optimistic concurrency control with eager locking and tracks conflicts at word granularity.
- **CX** [26]: Created with the CX universal construction, which produces wait-free operations and does not serialize read-only operations. It is the fastest wait-free universal construction we are aware of, although it still copies the entire data structure. We use the original authors' implementation [27].
- **COW** [5]: Created with a COW-based approach inspired by Ben-David et al., which produces lock-free writing operations and wait-free read-only operations, without serializing read-only operations. We implement COW ourselves.
- **Hand-Crafted:** We use hand-crafted designs of comparable algorithms, as we are not aware of concurrent implementations of exactly the same trees. We compare occ[RB] to SnapTree, a concurrent AVL tree with optimistic (lock-based) synchronization [11]. We compare occ[Radix] to a lock-free version from the same repository [79]. We compare occ[B+tree] to Brown's lock-free B-slack tree [12] (a B-tree variant). We use the original authors' implementations.

Our tree selection covers a spectrum of mechanically- and hand-crafted synchronization techniques. While some of these techniques do not form an "apples to apples" comparison with Occualizer's optimistic lock-based synchronization (e.g., due to being lock- or wait-free), the point is that they represent the space of currently available mechanical techniques.

Workloads. We populate the tree with 64 M uniformly random 8-byte keys and then run a workload for 3 seconds. Each workload has a different mix of operation types (Table 3). Our workloads are inspired by the standard Yahoo! Cloud Serving Benchmark (YCSB) [23] workloads, which are designed to simulate real-world application workloads, but differ in that (1) we replace updates with insertions and do not test range

Workload	Description
R-100	100% lookups (YCSB-C)
R-97	97% lookups, 3% insertions (≈ YCSB-B)
R-75	75% lookups, 25% insertions
R-50	50% lookups, 50% insertions (≈ YCSB-A)

Table 3: Contention workloads.

queries, as not all implementations support these operations; and (2) we use more levels of writing operations, for more fine-grained insight. Experiments run under SetBench [13], a benchmarking harness for concurrent C++ dictionaries that provides epoch-based memory reclamation.

Throughput. Figures 4–6 show the aggregate throughput for each workload with varying numbers of threads for the B+tree, radix, and red-black tree variants, respectively. All variants except Global-Lock scale on the read-only workload (R-100), although GCC-TM's throughput drops for RB at 24–28 cores. But for workloads with any level of writing operations, only Occualizer and the hand-crafted trees scale, with throughput of mechanically-crafted trees typically flatlining at low core counts.[5]

The mechanically-crafted trees do not scale due to suboptimal synchronization. In CX and COW, all writing operations are serialized, although serialization in COW, which is done with a CAS to the tree root, is significantly faster than in CX, where writing operation participate in a helping scheme and may copy the entire data structure. In GCC-TM, the problem is its conservative version of optimistic synchronization, which retries any transaction if any node it reads is updated before the transaction commits. Thus, for example, GCC-TM breaks down on RB—where tree rebalancing writes to the top of the tree—much earlier than on Radix.

Compared to the hand-crafted trees, Occualizer achieves comparable or better throughput when the level of mutation is low (R100-R97). The cases in which Occualizer is faster are due to differences in the underlying algorithms, which manifest when synchronization overhead is low. This effect demonstrates the power of Occualizer's approach, which removes the difficulty of adding synchronization to a tree from consideration, and thereby allows focusing on the (sequential) "quality" of the tree, i.e., how fast it is to search.

As mutations increase (R75-R50), however, synchronization becomes the dominating factor and Occualizer significantly underperforms the hand-crafted trees. The reason is that Occualizer writing operations are slower than in the hand-crafted trees, due to bookkeeping and copying overhead (see § 7.3), and so as the proportion of mutations grows, overall throughput degrades.

[5]The only exception is GCC-TM on Radix. In Radix, the tree only grows downwards, so any non-null pointer read during a traversal is immutable. GCC-TM thus rarely aborts transactions even with moderate mutation levels, and so achieves high throughput.

Tree	Throughput relative to hand-crafted	R-100	R-97	R-75	R-50
B+tree	Occualizer	0.79	0.77	0.64	0.72
	Best mech-crafted	0.82	0.38	0.05	0.05
	Gap shrink	**—**	**2.01**	**12.28**	**13.54**
Radix	Occualizer	1.08	0.84	0.73	0.50
	Best mech-crafted	1.15	0.90	0.74	0.36
	Gap shrink	**—**	**0.93**	**0.99**	**1.38**
RB	Occualizer	1.01	0.90	0.31	0.19
	Best mech-crafted	1.02	0.24	0.04	0.03
	Gap shrink	**—**	**3.78**	**8.64**	**6.08**

Table 4: Throughput difference between Occualizer and the best result of the mechanically-crafted trees at 28 cores, for each workload.

Tree	B+tree	Radix	RB
occ[·]	2.14 GB	2.10 GB	3.12 GB
Global-Lock	0.91×	0.93×	0.93×
GCC-TM	0.91×	0.96×	0.95×
CX	13.5×	13.18×	13.03×
COW	1.02×	1.18×	1.04×
Hand-Crafted	1.12×	0.97×	1.01×

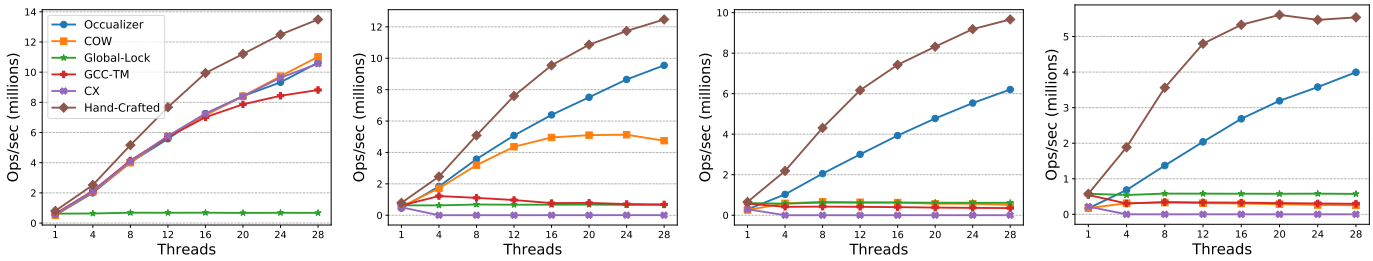
Table 5: Memory use at 28 cores (R-50), normalized to Occualizer's.

The takeaway is that Occualizer significantly shrinks the performance gap between mechanically- and hand-crafted trees in workloads with mutations. Table 4 reports this gap, and by how much Occualizer shrinks it. Overall, Occualizer shrinks the gap by up to 13.54×, 1.38×, and 8.64× for B+tree, Radix, and RB, respectively.

Memory use. Node copies made by an Occualizer tree may increase memory use compared to its sequential version, as a function of how quickly old nodes are reclaimed. To quantify this effect, Table 5 shows peak memory use for each tree in the R-50 workload (results for other workloads are similar). Both Occualizer and COW indeed use more memory than the sequential baseline (captured by Global-Lock), but Occualizer's LCOW increases memory use by 7%–9% whereas COW, which copies entire paths, adds an overhead of 11%–26%. In contrast, CX uses about 14× the memory of Global-Lock. The reason is that CX maintains multiple replicas of the data structure, so that read-only operations can read from a replica and avoid being serialized. Results of the hand-crafted algorithms are shown only for completeness, as they are not implementations of the same algorithm.

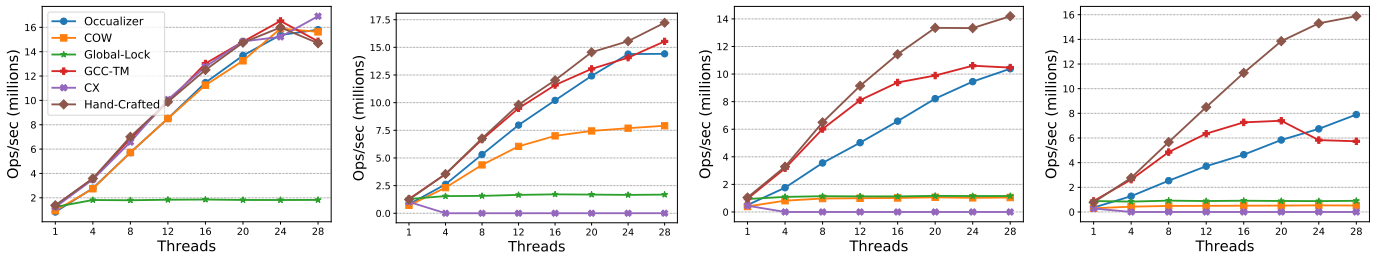
7.2 Full-system benchmark

We add occ[B+tree] as the index data structure in the STOv2 main-memory database system [58], and compare the result to the default index, Masstree [71], a hand-crafted concurrent



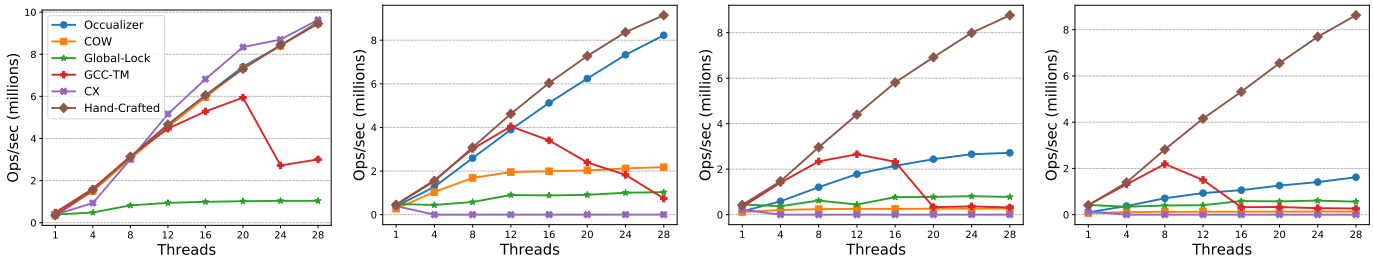
(a) R-100 (b) R-97 (c) R-75 (d) R-50

Figure 4: Throughput of B+tree variants for workloads with increasing amounts of mutation.



(a) R-100 (b) R-97 (c) R-75 (d) R-50

Figure 5: Throughput of radix tree variants for workloads with increasing amounts of mutation.



(a) R-100 (b) R-97 (c) R-75 (d) R-50

Figure 6: Throughput of RB (red-black tree) variants for workloads with increasing amounts of mutation.

tree combining aspects of B+trees and tries.[6]

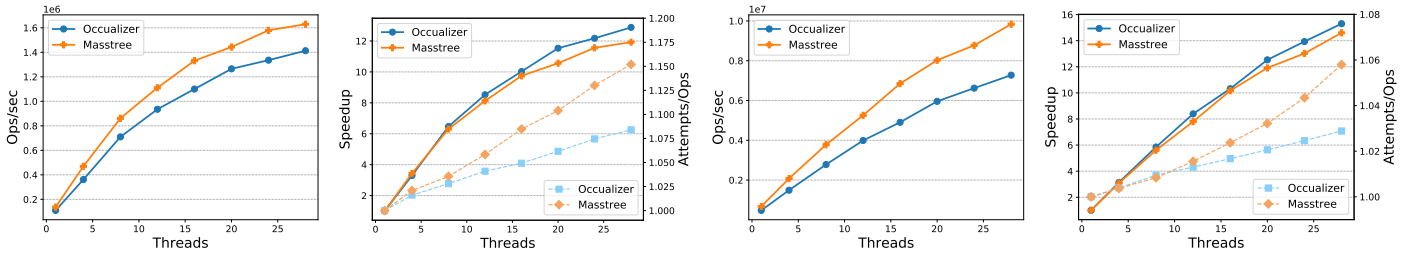
STO provides serializable transactions. Transactions are specified via C++ programs accessing STO's transactional interface. STO's transaction concurrency control is tightly coupled with Masstree, as it relies on Masstree node version numbers to detect and block certain anomalies. To integrate occ[B+tree] into STO, we therefore implement an equivalent versioning scheme in the `tlx` B+tree.

Workloads. We evaluate two transactional workloads, TPC-C and Voter. TPC-C is the industry standard benchmark for evaluating the performance of online transaction processing (OLTP) systems [28], by simulating an order processing application. We use a database with one warehouse, 100,000 items, and run the full mix of all TPC-C transactions. This workload performs index range queries. Voter is a benchmark that simulates a phone-based voting application. It consists of many short transactions and does not perform range queries.

Results. Figure 7 shows the throughput (committed transactions per second) and scalability of the system for both workloads, measured over a 10-second run. Scalability is measured by normalizing the throughput obtained with each index to the single-threaded throughput obtained with that index. On both workloads, occ[B+tree] is slower than Masstree, but has better scalability. As a result, the performance gaps between them shrinks as more threads are added: from a single-threaded difference of 22% and 29% for TPC-C and Voter, respectively, to a difference of 12% and 26% at 28 threads.

The reason behind occ[B+tree]'s better scalability is that Occualizer's optimistic synchronization protocol causes fewer operations to abort and retry than Masstree's protocol (which is also optimistic). Masstree uses per-node version counters to guarantee that searches observe only consistent node states. In Masstree's protocol, any operation—including a lookup—might abort and retry if its version checking indicates it may

[6] We use a B+tree here because it is closest algorithmically to Masstree, which is a B+tree variant. Comparing to, say, a red-black tree would not be meaningful, because Masstree outperforms a red-black tree for reasons unrelated to concurrency control (e.g., its much "shallower" tree structure).



(a) TPCC throughput. (b) TPCC speedup. (c) Voter throughput. (d) Voter speedup.

Figure 7: Throughput and speedup (throughput normalized to single-threaded throughput) of occ[B+tree] vs. Masstree in STOv2. Speedup plots also show the average number of attempts required to complete an operation (right Y axis).

Enabled methods	Relative throughput
None (call overhead)	93%
`occ_start`	87%
+ `occ_finish`	81%
+ `occ_get`	53%
+ `occ_set` = occ[B+tree]	40%

Table 6: Single-threaded throughput of occ[B+tree] relative to sequential B+tree (R-50 workload) as Occualizer's synchronization functionality is gradually enabled.

have observed an inconsistent node state. In contrast, Occualizer relies on LCOW to guarantee that observed nodes are consistent, and on the "forepassed" condition [44] to guarantee correctness of searches that traverse inconsistent tree states. In Occualizer's protocol, read-only lookups never abort and retry—only update operations do. As a result, as Figures 7b and 7d show (on the right Y axis), the average number of attempts required to complete an operation is larger in Masstree than in occ[B+tree]—and the difference grows with the number of threads.

7.3 Overhead analysis

To break down the sources of Occualizer's performance overhead, we evaluate the throughput impact of making the transformation but using a no-op implementation of each library method, then gradually adding in each method's actual implementation. We use single-threaded execution for this evaluation, because an Occualizer tree does not run correctly when any of the methods are disabled.

Table 6 shows the results, comparing occ[B+tree] to the original sequential B+tree, on the R-50 workload. The lion's share of overhead is due to `occ_get` and `occ_set`, which interpose on node field accesses. The impact of `occ_get` is $\approx 2\times$ that of `occ_get`, as reading fields is more frequent. Invoking the methods, `occ_start`, and `occ_finish` each degrade throughput by 6–7% points.

8 Conclusion

This paper presented Occualizer, a mechanical transformation for adding scalable optimistic synchronization to a sequential search tree implementation. Occualizer's specialization to trees enables designing a synchronization protocol that does not suffer from the limitations of transactional memory and universal constructions. Overall, Occualizer trees shrink the performance gap between these automatic transformations and hand-crafted trees by up to 13×.

Occualizer is limited, however, in that it applies only to sequential search trees that satisfy its prerequisites. Relaxing the prerequisites and automating the verification that an input tree satisfies them are interesting future directions, as is reducing the overhead of Occualizer's synchronization library. Our current Occualizer prototype also requires some manual steps to transform the input tree; automating these steps is an ongoing effort.

Occualizer's code is available at https://github.com/tomershanny/Occualizer.

Acknowledgements

We thank Yotam Feldman for many illuminating and enjoyable (often simultaneously) discussions. We thank the reviewers and the paper's shepherd, Irina Calciu, for their feedback.

This research was funded in part by the Israel Science Foundation (grant 2005/17) and the Blavatnik Family Foundation.

References

[1] The GNU Transactional Memory Library. https://gcc.gnu.org/onlinedocs/gcc-5.5.0/libitm.pdf, 2021.

[2] James H. Anderson and Mark Moir. Universal Constructions for Multi-Object Operations. In *PODC*, 1995.

[3] James H. Anderson and Mark Moir. Universal Constructions for Large Objects. *IEEE TPDS*, 10(12), 1999.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, 2012.

[5] Naama Ben-David, Guy E. Blelloch, Yihan Sun, and Yuanhao Wei. Multiversion Concurrency with Bounded Delay and Precise Garbage Collection. In *SPAA*, 2019.

[6] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder – A Transactional Record Manager for Shared Flash. In *CIDR*, 2011.

[7] Timo Bingmann. STX B+ Tree C++ Template Classes. https://panthema.net/2007/stx-btree, 2013.

[8] Timo Bingmann. TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers. https://panthema.net/tlx, 2018.

[9] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD '18*, 2018.

[10] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, MIT, 2014.

[11] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree. In *PPoPP*, 2010.

[12] Trevor Brown. B-slack Trees: Space Efficient B-Trees. In *SWAT*, 2014.

[13] Trevor Brown. SetBench: Powerful tools for data structure experiments in C++. https://gitlab.com/trbot86/setbench, 2021.

[14] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *OPODIS*, 2013.

[15] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-Box Concurrent Data Structures for NUMA Architectures. In *ASPLOS*, 2017.

[16] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB*, 2001.

[17] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD*, 2018.

[18] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A Universal Construction for Wait-Free Transaction Friendly Data Structures. In *SPAA*, 2010.

[19] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *ASPLOS*, 2012.

[20] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *EuroSys*, 2013.

[21] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *SOSP*, 2013.

[22] Douglas Comer. Ubiquitous B-Tree. *ACM CSUR*, 11(2), 1979.

[23] Brian F. Cooper, Adam Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC '10*, 2010.

[24] Jonathan Corbet. Trees I: Radix trees. https://lwn.net/Articles/175432/, 2006.

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. The MIT Press, 2022.

[26] Andreia Correia, Pedro Ramalhete, and Pascal Felber. A Wait-Free Universal Construct for Large Objects. In *PPoPP*, 2020.

[27] Andreia Correia, Pedro Ramalhete, and Pascal Felber. CX source code. https://github.com/pramalhe/CX, 2020.

[28] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, 2007.

[29] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS*, 2015.

[30] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, 2013.

[31] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *DISC*, 2006.

[32] Dave Dice, Nir Shavit, and Ori Shalev. Red-black balanced binary search tree. https://github.com/gramoli/synchrobench/blob/master/c-cpp/src/trees/rbtree/rbtree.c, 2006.

[33] Nuno Diegues and Paolo Romano. Self-Tuning Intel Transactional Synchronization Extensions. In *ICAC*, 2014.

[34] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *PACT*, 2014.

[35] Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal Constructions That Ensure Disjoint-Access Parallelism and Wait-Freedom. In *PODC*, 2012.

[36] Rob Ennals. Software transactional memory should not be obstruction free. Technical Report IRC-TR-06-052, Intel Research, 2006.

[37] Jason Evans. Scalable memory allocation using jemalloc. http://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919, 2011.

[38] Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue Adaptive Universal Constructions. In *DISC*, 2009.

[39] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. In *PPoPP*, 2012.

[40] Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-Efficient Wait-Free Synchronization. *Theory of Computing Systems*, 55(3), 2014.

[41] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[42] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic Transactions. In *DISC*, 2009.

[43] Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. Order out of Chaos: Proving Linearizability Using Local Views. In *DISC 2018*, 2018.

[44] Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. Proving Highly-Concurrent Traversals Correct. *PACMPL*, 4(OOPSLA), 2020.

[45] Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.

[46] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-Structured Data Stores. In *EuroSys*, 2015.

[47] Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, 2015.

[48] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2010.

[49] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic Transactional Boosting. In *PPoPP*, 2014.

[50] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *SPAA*, 2010.

[51] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13, 1991.

[52] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *SIGOPS OSR*, 26(2), 1992.

[53] Maurice Herlihy and Eric Koskinen. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *PPoPP*, 2008.

[54] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, 1993.

[55] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[56] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12, 1990.

[57] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-Aware Transactions for Faster Concurrent Code. In *EuroSys*, 2016.

[58] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. In *VLDB*, 2020.

[59] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In *SIGMOD*, 2013.

[60] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[61] Siddharth Krishna, Nisarg Patel, Dennis Shasha, , and Thomas Wies. Verifying Concurrent Search Structure Templates. In *PLDI*, 2020.

[62] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. Go with the flow: compositional abstractions for concurrent data structures. *PACMPL*, 2(POPL), 2018.

[63] Michael Larabel. Intel To Disable TSX By Default On More CPUs With New Microcode. https://www.phoronix.com/scan.php?page=news_item&px=Intel-TSX-Off-New-Microcode, 2021.

[64] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5), 2006.

[65] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE '13*, 2013.

[66] Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. A Constructive Approach for Proving Data Structures' Linearizability. In *DISC 205*, 2015.

[67] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-Tree for New Hardware Platforms. In *ICDE '13*, 2013.

[68] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *SOSP*, 2011.

[69] Linux. lib/rbtree.c, source code file of Linux 5.17. https://github.com/torvalds/linux/blob/v5.17/lib/rbtree.c#L35-L57, 2022.

[70] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *USENIX ATC*, 2012.

[71] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys*, 2012.

[72] Paul E. McKenney and John D. Slingwine. Read-copy update: using execution history to solve concurrency problems. In *PDCS*, 1998.

[73] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC*, 2010.

[74] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. In *PDSIA*, 1999.

[75] Darko Petrović, Thomas Ropars, and André Schiper. On the Performance of Delegation over Cache-Coherent Shared Memory. In *ICDCN*, 2015.

[76] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing Conflicting Transactions in an STM. In *PPoPP*, 2009.

[77] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC*, 1995.

[78] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *JACM*, 32, 1985.

[79] Hugo Sousa. RadixTree. https://github.com/ha2398/radix-tree, 2016.

[80] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, 2013.

[81] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[82] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*, 2006.

[83] Ziqi Wang, A. Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, M. Kaminsky, and D. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD '18*, 2018.

[84] Lingxiang Xiang and Michael Lee Scott. Compiler Aided Manual Speculation for High Performance Concurrent Data Structures. In *PPoPP*, 2013.

[85] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*, 2016.

Immortal Threads: Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers

Eren Yıldız
Ege University, Turkey

Lijun Chen
University of Trento, Italy

Kasım Sinan Yıldırım
University of Trento, Italy

Abstract

We introduce Immortal Threads, a novel programming model that brings pseudo-stackful multithreaded processing to intermittent computing. Programmers using Immortal Threads are oblivious to intermittent execution and write their applications in a multithreaded fashion using common event-driven multithreading primitives. Our compiler fronted transforms the stackful threads into stackless threads that waste a minimum amount of computational progress upon power failures. Our runtime implements fair scheduling to switch between threads efficiently. We evaluated Immortal Threads on real hardware by comparing it against the state-of-the-art intermittent runtimes. Our comparison showed that the price paid for the Immortal Threads is a runtime overhead comparable to existing intermittent computing runtimes.

1 Introduction

Advancements in low-power electronics and energy harvesters exploiting ambient sources (e.g., solar [20], indoor light [21], and radiofrequency [27]) paved the way for sustainable systems that can work without batteries. Recent studies have demonstrated promising examples of these systems, such as body implants [23] and long-lived wearables [51], where continuous power is not available and changing batteries is difficult. There are several microcontroller-based batteryless computing platforms (e.g., WISP [46], Flicker [24], Camaroptera [42] and Engage [16]) developed by the researchers. Instead of a battery, these platforms comprise a capacitor that powers all hardware components, including the ultra-low-power microcontroller (MCU), sensors, communication circuitry, and other peripherals. When a batteryless platform consumes the energy stored in its capacitor, it turns off due to a power failure. The platform charges its capacitor until the stored energy exceeds an operating threshold, which turns on the platform again. Therefore, the software on batteryless platforms runs *intermittently* due to frequent power failures and charge-discharge cycles.

Each power failure clears the CPU registers and the volatile memory during an intermittent execution. Hence, the computation might not progress forward since the control returns to the application's entry point [11]. Moreover, power failures may cause data stored in non-volatile memory to be partially updated, leading to memory inconsistency [43]. The prior art proposed mainly two approaches to overcome these issues. The first one is to place checkpoints in program source [6, 8, 26, 28, 30, 31, 33, 36, 44, 53], which store their continuation (i.e., the control state including the registers, stack and global data) in non-volatile memory. After a power failure, control resumes from the latest successful checkpoint location. Another approach is to employ a task-based programming model [10, 19, 25, 34, 37, 38, 45, 54], which requires programmers to implement their programs as a collection of tasks and transitions between them. This model eliminates the cost of checkpoints, since the all-or-nothing semantic of tasks, defined by the programming model, means that a function pointer to the current task is enough to represent the continuation of the program, which makes saving and restoring it from non-volatile memory extremely cheap [10].

Despite efficiency, the task-based model poses significant problems in developing *event-driven* applications [17, 32]. This situation prevents the widespread adoption of intermittent systems since most sensing applications are event-driven.

P1-Event Handling Complexity: Event handling, in general, is implemented in the form of state machines that require explicit management of states and transitions [18]. Implementing event-driven applications using the task-based model requires programmers to manage (i) task partitioning, (ii) task-based control flow, (iii) event states, and (iv) state transitions simultaneously. This situation creates an excessive cognitive burden concerning event-driven intermittent computing.

P2-Limited Concurrency: Existing event-driven task-based systems (e.g., [45, 54]) cannot fully support preemptive threads. Since tasks execute atomically, they voluntarily yield the control, and other tasks cannot preempt them. Moreover, tasks cannot block on events, trigger new threads of execution, and notify the completion of event processing. Therefore, pro-

grammers need to partition long-running computation (e.g., compression [29]) into a set of tasks to avoid missing events.

P3-Wasted Progress: Partial execution of tasks (due to power failures) leads to loss of computational progress within tasks since tasks have all-or-nothing semantics. This issue increases event response time, which is critical for event-driven systems. Recent work proposed loop continuation to preserve computational progress after each loop iteration by selectively *violating* the task-based model [22] (see Sections 1 and 6).

Problem Statement. Considering the mentioned problems, we seek a programming model that:

(Req.-1) removes the cognitive load of the task-based model while retaining its lightweight characteristics;

(Req.-2) brings the flexibility of preemptive multithreaded programming to intermittent systems;

(Req.-3) enables progress from the point where a thread has been interrupted due to a power failure.

Challenges. Fulfilling these requirements is not trivial. To satisfy (Req.-1), Kortbeek et al. [31] proposed giving up the task-based model and using lightweight and sparse checkpoints that save all registers and only the memory segments modified by the program. However, to fulfill (Req.3), checkpoints need to be placed almost at each line in the code. This situation creates an unmanageable overhead even with these lightweight checkpoints. Finally, concerning (Req.-2), we note that it is not enough to use a checkpoint runtime on top of an existing multithreaded OS, since OS primitives such as mutexes, semaphores, as well as interrupt handling, must be implemented taking the intermittence into account, to avoid memory consistency issues due to partial updates.

Contributions. In this paper, we introduce Immortal Threads that brings *pseudo-stackful* preemptive multithreaded programming model to event-driven intermittent computing. Programmers using Immortal Threads are oblivious to intermittent execution and write their applications in a multithreaded fashion using plain C without tasks (see Figure 1). Immortal Threads compiler fronted transforms stackful threads into stackless threads, inserts ultra-lightweight checkpointing mechanisms under the hood to minimize wasted progress, and maintains the memory consistency. The Immortal Threads library implements a preemptive scheduler to switch between threads and provides common event-driven primitives such as semaphores and blocking event wait operations. Our real-world experiments showed that Immortal Threads has runtime overhead comparable to the prior art intermittent runtimes InK [54], Alpaca [34] and TICS [31]. Moreover, during frequent power failures, Immortal Threads reduced execution time and wasted work by up to 40% and 90%, respectively.

In summary, Immortal Threads introduces the following contributions:

(1) Preemptive Multithreading: For the first time, we enable preemptive multithreading for event-driven intermittent systems, which provides programming flexibility and eliminates the cognitive burden of task-based programming.

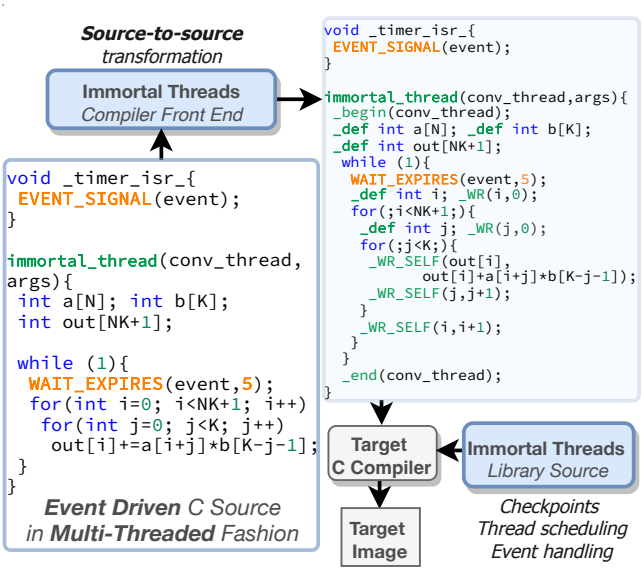


Figure 1: With Immortal Threads, programmers write applications in a multithreaded fashion without concerning intermittent execution, and focus **only** on *event-driven* aspects.

(2) Almost-Free Checkpoints: We propose a novel checkpointing technique, inspired by Dunkels et al. [18], that saves *only* the program counter rather than all registers and memory.

(3) Just-in-time Privatization: We propose a novel technique that eliminates the need for creating static versions of non-volatile program variables to keep memory consistent.

(4) Micro Continuation: Thanks to almost free checkpoints and just-in-time privatization, threads always progress from their latest memory update, and they do not waste computational progress upon power failures.

(5) Open-source Release: We release Immortal Threads as a C library with compiler support (via [55]) for the widespread adoption of intermittent computing.

2 Background and Related Work

Batteryless computing platforms comprise ultra-low-power MCUs with embedded non-volatile memory. For instance, MSP430FR5969 [48], one of the mainstream MCUs used in batteryless platforms, has 64kB of FRAM [50] and 2kB of SRAM memory. FRAM stores data that will persist upon power failures. The key challenges of intermittent computing are the loss of computational progress after power failures and memory inconsistency issues. Power failures reset the MCU, and the control returns to the application's entry point. Moreover, power failures might keep persistent variables (i.e., variables maintained in non-volatile memory) partially updated and in an inconsistent state. Code blocks with WAR

(Write-After-Read) dependencies on persistent variables are not idempotent, since they might produce different results when the MCU re-executes them after a power failure [43]. For example, assume that `x` is a persistent variable and the program executes `{x++; vector[x]=v;}`. A power failure after `x++` re-executes `x++` and leads `x` to be increased twice.

2.1 Intermittent Computing Approaches

The prior art focused on the forward progress and memory consistency aspects of intermittent execution but also considered the timeliness of data processing and event-driven concurrency.

Checkpoints. In energy-guided checkpointing, the device continuously monitors the capacitor to perform a checkpoint on imminent power failure, for example, as in Hibernus [6]. However, voltage monitoring is quite expensive in terms of energy consumption [52]. In software-only checkpointing (e.g., DINO [33], Chinchilla [36] and TICS [31]), the program source is instrumented with checkpoints, either by a programmer or a compiler. The checkpoints are double-buffered in non-volatile memory to prevent the latest consistent checkpoint from being superseded immediately by an inconsistent (i.e., partially updated) checkpoint. Moreover, a compiler analysis is required to determine the modified persistent variables between two checkpoints and create their versions to prevent violations of idempotency upon resumption [33]. After a power failure, the checkpointing runtime restores the versions that isolate the code from partially updated versions, and the control resumes from the latest successful checkpoint location. There are several other works that aim to reduce the overhead of checkpoints [3, 4].

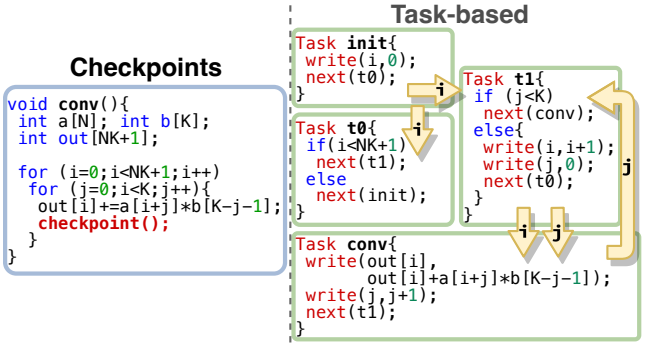


Figure 2: The task-based and checkpoint-instrumented versions of a 1-D convolution code. Arrows among the tasks denote channels that hold versions of task-shared variables.

Task-based Model. This model eliminates the overhead of checkpoints by proposing tasks that do not have a restoration cost. Tasks have read-only inputs and write-only outputs (called channels [10]), which are maintained in non-volatile memory separately. Tasks are inherently idempotent since separate channels avoid WAR dependencies in the task body. Runtimes execute tasks atomically with all-or-nothing semantics. The task-based model employs static multi-versioning by creating multiple copies of the data distributed over the inputs and outputs of the tasks (see Figure 2). Alpaca [34] avoided multi-versioning by proposing *privatization* that creates local copies of the task-shared persistent variables. Each task loads its local copies with the original values, manipulates local copies, and commits them to the original locations upon completion.

Timely Execution. Data (processing) might expire due to charging times during intermittent execution. Mayfly [25], InK [54] and CatNap [37] proposed extensions to the task-based programming model to express timely data constraints and time-critical code. TICS [31] added extensions to checkpointing systems to enable timely data processing.

Event-driven Intermittent Computing. InK [54] proposed task threads, which are triggered by events to execute a sequence of tasks. Coati [45] handles the event-driven concurrency issues by serializing concurrent interrupts with the tasks to keep the shared persistent state consistent. CatNap [37] isolated energy for reliable intermittent execution of periodic events, which are time-critical tasks in a task-based model. TICS [31] does not support event-driven constructs, but it can checkpoint event-driven legacy code—though these checkpoints do not guarantee the semantically correct operation [31] (see Section 5.1).

2.2 Embedded Concurrency Models

Several studies proposed different concurrency models for embedded systems [2, 7, 29, 32, 32, 40, 41]. We classify the main differences in these models into two spheres: whether the concurrency is cooperative or preemptive, and whether the concurrency units (i.e., "threads") are stackful or stackless.

Concurrency Approaches. Stackful concurrency has programming expressiveness, since continuations can be set anywhere in the thread's call stack by preserving the local variables [29]. It is costly since each thread requires its own stack, and thread preemption requires storing all registers into the continuation. In stackless concurrency, threads share the same stack, and local variables are not maintained when a thread blocks. As an example, Protothreads [18] implements a stackless cooperative concurrency model. Consequently, a protothread can yield control only within its main body, not within the body of a function it calls.

Concurrency and Checkpoints. The former sets continuations (a.k.a. execution context) to switch context among threads, while the latter sets them to restore the program state after reboot. Therefore, from the perspective of concurrency, software-only checkpointing systems are non-preemptive, in the same way stackful threads voluntarily set a continuation and yield control. Besides, energy-guided checkpointing systems are preemptive since they stop thread execution and set a

Intermittent Runtimes	Main Features							Timely, Event-Driven Intermittent Program Development
	Task-based or Checkpointing	*Run-time Overhead*	*Event-driven Support*	*Cognitive burden*	*Lost Work*	*Micro Continuation*	*Timely Execution*	
Dewdrop [9], Mementos [44], DINO [33], HarvOS [8]	Checkpointing	High ✗	No Support ✗	Low ✓	Low to High ✗	No ✗	No ✗	N/A ✗
Ratchet [53]	Checkpointing	Very High ✗	No Support ✗	Low ✓	Very Low ✓	No ✗	No ✗	N/A ✗
Chinchilla [36]	Checkpointing	Medium ✗	No Support ✗	Low ✓	Low ✓	No ✗	No ✗	N/A ✗
Chain [10], Coala [38], Alpaca [34]	Task-based	Low ✓	No Support ✗	High ✗	High ✗	No ✗	No ✗	N/A ✗
Mayfly [25]	Task-based	Low ✓	No Support ✗	High ✗	High ✗	No ✗	Yes ✓	N/A ✗
TICS [31]	Checkpointing	Medium ✓	No Support ✗	Low ✓	High ✗	No ✗	Yes ✓	N/A ✗
InK [54], Rehash [5]	Task-based	Low ✓	Limited Support ✓	High ✗	High ✗	No ✗	Yes ✓	Difficult ✗
Coati [45]	Task-based	Low ✓	Limited Support ✓	High ✗	High ✗	No ✗	No ✗	Difficult ✗
CatNap [37]	Task-based	Low ✓	Limited Support ✓	High ✗	High ✗	No ✗	Yes ✓	Difficult ✗
Immortal Threads (this work)	**Checkpointing** (almost zero overhead)	**Low** ✓	**Full Support** ✓ (multithreading)	**Very Low** ✓ (almost zero)	**Very Low** ✓ (almost zero)	**Yes** ✓	**Yes** ✓	**Easy** ✓(almost the same as in continuous systems)

Table 1: A comparison of the main features of Immortal Threads with the relevant intermittent computing approaches.

continuation upon an imminent power failure. Intuitively, the task-based model is a form of static non-preemptive stackless checkpointing system. Static and non-preemptive because task decomposition is done at programming time and checkpoints are taken only at task boundary, and stackless because only the active task's function pointer is checkpointed. Similar to stackless threads, the low-overhead of the task-based model comes at the cost of imposing a programming model with a high cognitive load.

2.3 Drawbacks of Prior Works

Table 1 presents a comparison of the main characteristics of this work and the existing intermittent computing approaches.

1- Event-handling Complexity with Tasks. The task-based implementation of a small deep neural network (DNN) inference in Gobieski et al. [22] has 18 tasks and 61 control flow declarations. Implementing an event-triggered state machine using tasks is even more complex. For example, a low-level radio driver depicted in Dunkels et al. [18, Table 1] has 26 explicit states and 32 state transitions. Implementing this driver using existing task-based event-driven intermittent runtimes [37, 45, 54] requires handling task partitioning and control flow, states, and transitions simultaneously, which is an unmanageable cognitive load.

2- Programming Model Violations. Power failures lead to the waste of computational progress (and energy) when they prevent the execution from reaching the successive checkpoint or the end of the current task. For example, a power failure in the middle of the convolution task while performing the DNN inference in Gobieski et al. [22] might lead to the loss of almost 150000 multiplications. Prior work proposed loop continuation [22] that avoids wasted work by allowing tasks to directly modify non-volatile memory in a loop nest, which is a violation of the task-based model.

3- Limited Concurrency. None of the existing intermittent systems supports the stackful preemptive concurrency model. As Yildirim et al. [54] comments, checkpointing an existing preemptive multi-threading operating system is not practicable due to the inefficiency issues and the memory inconsistencies caused by intermittence-unaware interrupt handling. Similar concerns hold for existing works (e.g., [41]) that can transform stackful threads into stackless continuations for continuously powered systems. For the sake of efficiency, many existing work on intermittent computing utilizes a lightweight stackless cooperative concurrency approach via tasks [37, 45, 54].

3 Immortal Threads: Overview

Immortal Threads consists of a programming interface, a compiler frontend, and a small run-time library, which bring *pseudo-stackful* preemptive multithreading model into intermittent computing. Programmers using Immortal Threads are oblivious to intermittent execution, and they develop their programs in a multithreaded fashion as they are programming a continuously powered system. The compiler frontend transforms the source code into stackless continuations that handle intermittency without programmer intervention.

As depicted in Figure 1, the main building block of an intermittent event-driven application is the thread of execution that continues running from where it left upon power failures, which we call *immortal thread*. Unlike the task-based model (which requires explicit idempotent code generation via task splitting), the unnecessary details of the intermittent execution are not visible to the programmers. The duties of the programmers are to (i) identify the events in their system, (ii) design their systems as a set of threads that are the handlers of these events, (iii) manage the necessary state management and state transitions, and (iv) consider timing aspects during event-driven intermittent execution. It is worth mentioning that duties (i)–(iii) are identical to the steps followed to develop event-driven applications in continuously powered systems [17, 32]. Differently, in (iv), programmers embed (if required) the necessary program logic to check event expiration due to the delays stemming from the charge/discharge cycles during the intermittent execution.

Language Construct	Explanation
`_SEM_WAIT(sem)/_SEM_POST(sem)`	wait on/post semaphore **sem**
`_SEM_POST_ISR(sem)`	post semaphore **sem** in an ISR
`_EVENT_SET_TIMESTAMP(e, t)`	sets the timestamp of **e** as **t**.
`_EVENT_GET_BUFFER(e)`	returns a pointer to the data buffer of **e**
`_EVENT_GET_TIMESTAMP(e)`	returns the timestamp of **e**
`_EVENT_WAIT(e,buf)`	blocking wait on **e** w/o expiration time, returns the event data via **buf**
`_EVENT_WAIT_EXP(e, buf, t)`	blocking wait on **e** w/ expiration time **t**, returns the event data via **buf**
`_EVENT_SIGNAL(e)`	signals the event and unlocks the thread waiting on the event

Table 2: Immortal Threads core language constructs.

Language Construct	Explanation
`_begin(name)/_end(name)`	immortal body start/end
`_def/_gdef`	pseudo local variables and persistent global variables
`_WR(arg,val)`	**arg** = **val** (variable assignment operations w/o W-A-R, e.g., `x=5`)
`_WR_SELF(type,arg,val)`	**arg** = **(type) val** (variable assignment operations w/ W-A-R, e.g., `x++`)
`_call(name,...)`	call immortal function **name** with appropriate arguments

Table 3: Main interfaces used by the compiler frontend.

3.1 Programming Model

Immortal Threads supports the common multithreaded event-driven language constructs, as presented in Table 2.

Timely Events and Blocking Wait. Immortal Threads provides an event primitive that builds a bridge between threads and ISRs (interrupt service routines). Threads can block (i.e., wait) on events using `_EVENT_WAIT` and `_EVENT_WAIT_EXP` interfaces, which suspend threads until the relevant event occurs. Signaling an event via the `_EVENT_SIGNAL` interface unblocks the waiting thread to continue its execution. Immortal Threads cannot guarantee event handling deadlines, but programmers can provide an expiration time to catch outdated events and prevent unnecessary event processing. To detect event expiration, programmers can use `_EVENT_WAIT_EXP`, which subtracts the current time from the timestamp of the event. This interface unblocks the corresponding thread if the result of the subtraction is less than the expiration time provided by the programmer. Blocking wait interfaces also pass a pointer to the event data to let the waiting thread copy these data into its thread-local buffer.

Wait and Post Semaphores. Immortal Threads provides a binary semaphore implementation for inter-thread signaling. A thread can block (wait) on a semaphore using the `_SEM_WAIT` interface. Another thread can post this semaphore using `_SEM_POST` interface to unblock that thread. ISRs can also post semaphores using a separate interface `_SEM_POST_ISR`.

ISRs and Event Signaling. In Immortal Threads model, interrupts have *all-or-nothing* semantics. ISRs interface with the hardware, obtain the data, and deliver it to threads. Each ISR has an associated `event` structure. When an interrupt (i.e., an event) occurs, the ISR obtains a pointer to the event data buffer via the `_EVENT_GET_BUFFER` interface. ISRs store the event data (e.g., the sensor reading) into this buffer and set the event timestamp via `_EVENT_SET_TIMESTAMP`. ISR commits these changes atomically and notifies the waiting thread via the `_EVENT_SIGNAL` interface. A power failure up to this point might lead to an event loss. Otherwise, the notified thread will obtain the event data and perform the necessary processing.

```
_immortal_thread(thread1){
 /* the main thread body */
 _begin
 /* variable declerations */
 _def uint8_t cnt;
 ...
  while(1){
   _EVENT_WAIT(event);
   /* necessary actions */
   _WR_SELF(uint8_t,cnt,cnt+1);
   ...
   /* notify thread 2 */
   _SEM_POST(sem);
   ...
  }
 _end
}
```

```
void _timer_isr_{
 /* notify event */
 _EVENT_SIGNAL(event);
}
```

```
_immortal_thread(thread2){
 /* the main thread body */
 _begin
 ...
  while(1){
   _SEM_WAIT(sem);
   /* necessary actions */
   _call(other_function);
   ...
  }
 _end
}
```

Figure 3: Output of the compiler frontend. Initially `thread1` and `thread2` are blocked. The timer ISR signals the event and unblocks `thread1` that unblocks `thread2`.

3.2 Execution Model and Multi-threading

Immortal Threads employs a multithreaded preemptive execution model by implementing a simple but efficient Round-Robin scheduling mechanism. Threads are initially blocked on events, waiting for ISRs to signal them. When an interrupt is triggered, the corresponding ISR signals an event, and the event handler thread wakes up and performs the computation. Therefore, there might be several threads running simultaneously during the execution of programs. Thanks to the compiler frontend, Immortal Threads manages the forward progress and memory consistency without programmer intervention.

3.3 Pseudo-Stackful Threads

The Immortal Threads compiler frontend performs a source-to-source transformation to convert the stackful threads into stackless continuations by employing almost-free checkpoints and just-in-time privatization. To do so, it uses the interfaces in Immortal Threads library (see Table 3). After the compiler pass, the transformed source code is linked with the Immortal Threads library. Figure 3 presents the output of the compiler frontend for a multithreaded event-handling example.

The compiler frontend instruments all programmer-defined functions, including thread entry points, to create immortal

functions. More specifically, an immortal thread is a concurrency unit whose entry point is an immortal function.

Instrumentation of an Immortal Function. The compiler frontend instruments all local variables by using `_def` followed by the data type and name (i.e., the ordinary way of variable declaration in C language). This operation converts programmer-defined local variables to persistent static variables with local scope. Compiler frontend instruments variable manipulations using `_WR` and `_WR_SELF` interfaces to ensure memory consistency. These interfaces manage WAR dependencies, perform checkpoints, and keep functions idempotent. `_WR` manipulates variables when the update operation does not include any WAR dependency. `_WR_SELF` manipulates variables when there is a WAR dependency during the update operation. For example, the Immortal Threads library implements the assignment `{x=0}` using `_WR(x,0)` since there is no WAR dependency during this update. For `{x=x+5}`, the necessary operation becomes `_WR_SELF(uint32_t,x,x+5)` since the variable `x` is read first and then written. Immortal Threads provides different interfaces for variable manipulations with WAR dependencies to implement *just-in-time privatization*, which we will explain in Section 4. Additionally, calls to other immortal functions in an immortal function body are instrumented with the `_call` interface, which makes setting micro-continuations inside called immortal functions possible. Finally, the compiler frontend also instruments the function body by wrapping it using `_begin/_end` block. When a thread starts running for the first time, the first instruction in its entry immortal function is executed. If a power failure interrupts thread execution, the thread continues from the last checkpoint performed by the underlying Immortal Threads runtime, which can also be deep down in the call stack.

Thread Preemption. Unlike common preemptive models, where the continuation is saved on preemption, Immortal Threads saves the continuation (i.e., checkpoint) on each memory update. This guarantees the idempotence of the execution until the next checkpoint. Therefore, the scheduler can simply interrupt the execution of a thread and switch to the other one.

4 Implementation of Immortal Threads

We implemented Immortal Threads library mainly using standard C macros and preprocessor directives. The library also includes functions for system initialization and scheduling operations. We implemented the source-to-source transformation using the LLVM & Clang LibTooling library [1].

Target Hardware. The current implementation of Immortal Threads library targets MSP430FR5994 [48] microcontroller from Texas Instruments that is equipped with 256KB FRAM and 8KB SRAM memory. Immortal Threads library uses a persistent time circuitry (which keeps track of time across power failures [14, 15]) to handle events and data expiration. It is worth mentioning that a persistent timekeeper is not a mandatory requirement for Immortal Threads runtime since it will work even without checking event time constraints. On the other hand, de facto intermittent computing platforms, e.g., Flicker [24], already include a persistent timekeeper circuit.

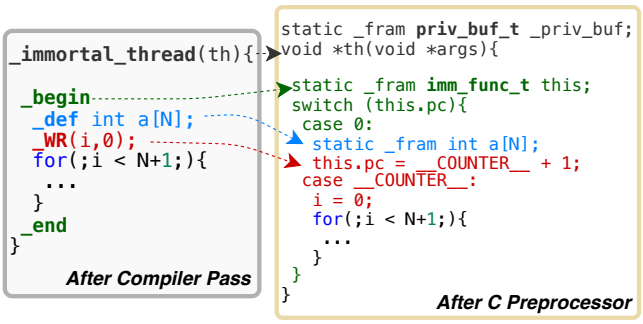


Figure 4: The structure of a source file after C preprocessor.

4.1 Immortal Function and Threads

Each immortal function (which can also be the entry point of a thread) maintains `imm_fn_t` structure that comprises a program counter (`pc`) to enable *micro continuations*, and a pointer to the same structure (`callee`) for calling other functions (via `_call`). For the *just-in-time privatization* operations, a privatization buffer (represented by `priv_buf_t`) is also maintained. All local variables are allocated in non-volatile memory as variables with static storage duration, which makes immortal threads (based on immortal functions) stackless. Figure 4 presents a sample output of the C preprocessor, which depicts how the privatization buffer (`__priv_buf`) and the function structure (`this`) are allocated in non-volatile memory.

4.2 Enabling Micro Continuations

Threads can be interrupted at any time by power failures, and their execution continues from the latest checkpoints. Immortal Threads library performs an almost-free checkpoint at each memory update via the `_WR` and `_WR_SELF` interfaces. Memory updates that lead to WAR dependencies (i.e., `_WR_SELF`) require *just-in-time privatization* to keep memory consistent.

Almost Free Checkpoints. Figure 4 presents how `_begin/_end` blocks (which are just C macros) are transformed into `_switch/_case` structures in C. Since the `imm_fn_t` structure for each immortal function is statically allocated, the `pc` field of is initialized with zero. Therefore, a function initially starts by executing its first case block `case 0:`. The almost-free checkpoint is just adding a new case statement at compile-time and modifying `pc` of the function. We implemented almost-free checkpoints using the standard GNU C preprocessor macro `__COUNTER__`, whose value increments each time the preprocessor encounters it. We implemented the core checkpoint code as follows:

```
#define _CP() \
    this.pc = __COUNTER__ + 1; case __COUNTER__:
```

If the thread (in which the function is running) restarts, the execution will continue from the `case` statement of the last checkpoint. However, only checkpoints are not sufficient to keep memory consistent. As indicated previously, `_WR` performs single-memory updates that do not lead to WAR dependencies. However, a sequence of operations `_WR(x,y);` `_WR(y,z);` form a WAR dependency. Therefore, we need to take a checkpoint before each memory update operation using `_WR`. We implemented this macro as follows:

```
#define _WR(arg,val) _CP(); arg=val;
```

Just-in-Time Privatization. Single memory updates that include WAR dependencies, e.g., `x++`, require a *two-phase commit* operation to keep memory consistent. In the first phase, Immortal Threads library creates a *private* version of the variable in the privatization buffer (`__priv_buf`) and updates the private version. Then a checkpoint is taken. In the second phase, Immortal Threads library commits the private version to the original variable. We present the implementation of the `_WR_SELF` macro that captures these steps below:

```
#define _WR_SELF(type,arg,val) \
    _CP(); *((type*)&__priv_buf.buffer)=val;\
    _CP(); arg=*((type*)&__priv_buf.buffer);
```

Thanks to just-in-time privatization, Immortal Threads does not require a compiler analysis to detect idempotent code blocks, as in Woude et al. [53]. Furthermore, there is no need for *static versioning*, as in Colin et al. [10]. Immortal Threads library forms a continuous sequence of idempotent code blocks by connecting them using almost-free checkpoints on the fly.

Calling other functions. When there is a power failure while a callee executes, the control should resume from the last memory update in the callee body. To call an immortal function, Immortal Threads library first checkpoints, saves the pointer to the callee in the caller's `_imm_fn_t` structure, and then makes the call as shown in the following pseudo-code:

```
#define _call(name,args) \
  _CP(); this.callee=name(args); \
  _CP(); this.callee->pc = 0; _CP();
```

If the callee successfully returns, the caller sets the program counter (`pc`) of the callee to zero and checkpoints. Consequently, the function will be able to be called again. If there is a power failure before the callee returns, the thread execution will restart from the entry immortal function, which will perform a set of nested function calls to reach the callee that has not finished yet. Therefore, the execution resumes from the last memory update in the leaf callee's body.

```
__asm__ volatile (
 "MOVA SP, %0" /* save SP on __sp variable */
 : "=m" (__sp) /* output */);

while (1) {
 functions[__th](0); /* call thread */
 __asm__ volatile (
  "ISR_return: \n" /* _schedISR will jump here */
  "MOVA %1, SP \n" /* restore old stack */
  "INC.B %0 \n"    /* __th++ */
  "CMP.B %2,%3 \n" /* if (__th == size) */
  "JNZ cont \n"
  "CLR.B %2 \n"    /* __th = 0 */
  "cont: \n"
  : "=m" (__th)    /* output */
  : "m" (__sp), "m" (__th), "m" (size)/* input */);
}

void __interrupt(TIMER0_A0_VECTOR) _schedISR(void){
 /* write return address ISR_return */
 __asm__ volatile ("MOV.W #ISR_return, 2(SP)");
}
```

Figure 5: The Round-Robin scheduler, which is the only platform-specific code in Immortal Threads library.

4.3 Thread Scheduling Implementation

The Immortal Threads scheduler includes platform-specific assembly code that switches the execution from the current thread to the next one. Figure 5 presents a part of our scheduler implementation. When the system restarts, the value of the stack pointer is saved in the variable `__sp`. The array `functions` contains pointers to the thread entries that are ready to run. The while loop indexes the threads with the persistent variable `__th` and calls them in order.

When the periodic timer of the scheduler fires, it interrupts the current thread, and the execution jumps to the `_schedISR` ISR routine. This ISR modifies the stack to replace the interrupt return address with the address of the label `ISR_return` in the scheduler loop. Upon interrupt return instruction (ISR routines execute `iret` upon return), the execution jumps to `ISR_return` label. At this point, the stack pointer is restored (using the old stack pointer in `__sp`) to continue the execution of the scheduler loop using its stack frame. Then, the value of the index `__th` is incremented, and the corresponding thread function is called. It is worth mentioning that thanks to micro continuation, the interrupted thread will not be in an inconsistent state. When the scheduler loop starts running again, it will continue execution from its latest checkpoint. Indeed, the ISR interruption acts as an artificial power failure.

Semaphores, Events and Data Races. Power failures might break the atomicity of operations on semaphores and events (e.g., post and wait) and might lead to data races. For example, if a thread modifies the semaphore but does not checkpoint due to a power failure, it will post the semaphore again after recovery. This situation leads to incrementing the semaphore twice. To prevent such issues, Immortal Threads library im-

plements event and semaphore operations by employing two-phase commit and double buffering, which are the main techniques proposed in the prior art to keep memory consistent despite power failures [31,34,54]. These operations firstly update the temporary values dedicated to events and semaphores. Then, the temporary values are atomically committed to their original locations. Upon system reboot, the scheduler checks if there is an uncommitted semaphore or event operation. If this is the case, it commits this operation. Then, it enables the interrupts and starts executing the threads. Immortal Threads library manages the data races between ISRs and threads by employing the same approach. Each event has a double-buffered event data buffer. An ISR does not modify the original buffer and immediately overwrites the event data. It uses the temporary buffer and then atomically commits it using a two-phase commit operation. These operations prevent the data races and inconsistency issues.

4.4 Compiler Frontend Implementation

We implemented Immortal Threads compiler frontend using the LLVM & Clang LibTooling framework. The AST produced by Clang is generally immutable, and source code rewriting cannot be directly reflected on the AST and its associated metadata. Therefore, it is necessary to keep track and manage the position offsets introduced by the transformations and solve conflicts when these transformations overlap. This limitation of Clang libraries, combined with the relative complexity of the entire source transformation for Immortal Threads, led us to adopt a multi-pass architecture inspired by the LLVM IR Pass framework. Each pass matches some parts of the AST and performs the appropriate source code rewriting. The rewritten source code is used to generate a new AST, on which the next pass operates.

Syntax Decomposition. One of the main challenges for source-to-source transformation is the `switch` constructs used in Immortal Threads lightweight checkpoints, which allow checkpoints only with a statement granularity. However, C programs might include expressions with WAR dependencies inside them, so it must be possible to perform checkpoints inside expressions. To this end, we decomposed these syntax constructs into separate statements so that it is possible to perform Immortal Threads lightweight checkpoints. In doing so, we paid special attention to aspects such as operator precedence and short-circuit evaluation.

Pessimistic Privatization due to Aliasing. The `_WR_SELF` interface, which performs JIT privatization, must be used for assignments where the left-hand side operand aliases with the right-hand side operand. In general, it is not possible to deduce all such aliases at compile time, e.g., when pointers are involved. Our current implementation pessimistically uses `_WR_SELF` instead of `_WR` when at least one operand contains a pointer dereference. We left integrating more advanced aliasing analysis as future work.

Shim API replacement. While a significant portion of the Immortal Threads operations is hidden from the programmer by the compiler frontend, primitives such as semaphores, mutexes, etc. (see Table 2) are visible to the programmer. These primitives have C macro-based implementations that generate `_case` statements for `_switch` blocks, which are inserted by the compiler frontend via `_begin` and `_end` statements later. Therefore, Clang fails to generate the initial AST that sets off the transformation pipeline. We solved this issue by providing these primitives as shim functions, and the compiler frontend replaces them with their actual macro-based implementations.

Pass Grouping Optimization. While the compilation time of the C language by modern compilers such as Clang is generally fast enough, having to re-parse the translation unit after each transformation is still a noticeable overhead when long source files are involved. Some of the presented passes depend on others. For example, instrumenting assignments with `_WR` and `_WR_SELF` is easily performed once syntax decomposition is done. On the other hand, some passes operate on orthogonal elements of the AST, for which we don't need to worry about source rewriting conflicts. We grouped these passes and executed them using the same AST.

Compiler Directives and Code Optimization. In exceptional cases, the programmer can modify the behavior of the Immortal Threads compiler using custom attribute directives (`__attribute__`). For example, we allow the programmer to mark idempotent functions so that they are not instrumented for the sake of some manual optimizations. This feature reduces the overhead of frequent checkpoints but creates a risk of wasted work. In Section 6 we discuss ways to improve this aspect. Furthermore, we also implemented a specific compiler optimization to coalesce successive `WR` and `WR_SELF` macros in the code to eliminate frequent checkpoints that might degrade the execution time of time-sensitive computational loads. The programmer can enable this optimization by passing a flag to the compiler frontend. In this case, the compiler puts the best effort to reduce the number of checkpoints in basic blocks. In summary, Immortal Threads compiler frontend enables the developers to select the trade-off between the checkpointing overhead and wasted work based on the specific requirements of their applications.

Switch Statements. We allow programmers to use a subset of the `switch` statement in their code (unlike Protothreads, which does not permit programmers to use switch statements). Specifically, we support switch statements in which all the statements associated with case labels either finish with a `break` statement or are empty, i.e., the case directly falls through to the next case. Given the constraint we put, it is straightforward to transform such use of the switch into an equivalent if/else based code. The compiler frontend terminates with an error message if it encounters an unsupported usage of the switch statement.

Function Reuse. Immortal Threads needs to create different instances of thread-shared immortal functions to prevent data

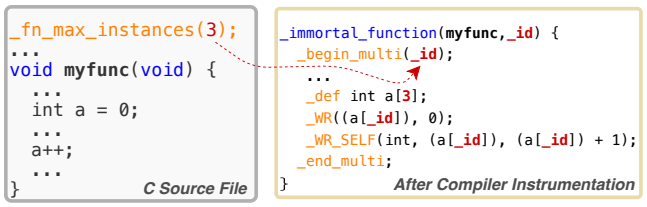


Figure 6: The compiler instrumented version of a sample function `myfunc` that is shared among multiple immortal threads.

races and memory inconsistencies. We implemented function reuse through a combination of compiler and Immortal Threads library support. Figure 6 presents a sample function that is shared among several immortal threads, and its instrumented version. The programmer uses a compiler directive (`_fn_max_instances` as indicated in the figure) to declare the maximum number of concurrent callers for the shared functions in the application. If the number of instances is not provided, the compiler can also use a default number to avoid programmer intervention. Our compiler modifies the signature of each shared immortal function by prepending an id parameter. Moreover, the compiler also transforms all local variables into arrays whose lengths correspond to the number of instances. Thus, each access to any local variables becomes an array access, where the index is the id. Alternatively, to avoid the overhead of accessing the array, the Immortal Threads compiler can also create copies of the same immortal function at the source code level. Therefore, it can replace the original immortal function's body with a call table that calls the appropriate function copy depending on the id parameter. This support lets the developer trade executable size for runtime efficiency. Besides, for each shared immortal function, the compiler allocates an associated metadata data structure containing a bitmap to represent unused instances, where each bit that is one represents a free instance. We present the pseudo-code of the macro that is used for calling shared functions as follows:

```
#define _call_multi(name,args) \
  _CP(); get_instance_id(&this->callee_id); \
   this.callee=name(this->callee_id, args); \
  _CP(); release_instance_id(&this->callee_id); \
   this.callee->pc = 0; _CP();
```

The caller of an immortal function must first get a free instance, that is, access the bitmap and clear a bit that is set (using `get_instance_id`). It is worth mentioning that not getting a free instance should not happen by design. The programmer must ensure to provide the correct `_fn_max_instances` number configuration. If no free instance is available at runtime, it's an assertion failure. Once the immortal function returns, the caller must release the called immortal function instance by setting the previously cleared bit (using `release_instance_id`). As a side note, recursive functions are not supported in the current implementation of Immortal Threads. We argue that this is not a significant limitation, as recursion is generally avoided in embedded systems.

5 Evaluation

We proceed with the evaluation of Immortal Threads by presenting a performance comparison against three state-of-the-art runtimes Alpaca [34], InK [54], and TICS [31].

Benchmarks. We selected Bitcount (BC), Cuckoo Filter (CF), and Activity Recognition (AR) as the main benchmarks since they are widely used in previous works [31, 34, 54]. We also considered the DNN inference presented in Gobieski et al. [22] as a benchmark since the inference operations are computationally intense (e.g., the first convolution layer requires 150080 multiplications) and access non-volatile memory excessively (FRAM is more expensive compared to SRAM access). We used the BC, CF, and AR implementations in publicly available code repositories of Alpaca, InK, and TICS during our evaluations. We also considered the publicly available plain C versions of these benchmarks, which we call *Plain-Ram*, where all variables are in SRAM (no FRAM access). Therefore, they do not have overheads regarding non-volatile memory access, checkpoints, privatization, etc. Moreover, we created the *Plain-Fram* versions of these benchmarks where all variables are maintained in FRAM. Therefore, they have an additional FRAM access overhead compared to Plain-Ram versions. Note that the Plain-Ram and Plain-Fram implementations do not guarantee forward progress and memory consistency.

Compiler Directives. In the task-based implementations of the benchmarks (in Alpaca and InK), we observed that some functions are not declared as tasks to reduce task transition overheads and employ a *manual* compile-time optimization for these task-based systems. These functions are idempotent since they do not modify their inputs or global variables. Moreover, they are mostly small in size and frequently called at runtime. Similarly, in Immortal Threads implementations of these benchmarks, we annotated these idempotent functions (using the compiler directives mentioned in Section 4.4) to bypass unnecessary compiler instrumentation and reduce the number of checkpoints and the execution time overhead of Immortal Threads. Using annotations, we marked eight functions in BC, six functions in CF, three functions in AR, and seven functions in DNN, respectively. Furthermore, for the DNN benchmark, we enabled the checkpoint coalescing feature of our compiler frontend to reduce the memory access overhead of the data-intensive computations. These features of Immortal Threads compiler frontend allowed us compile-time optimizations *without programmer involvement* (excluding the annotation of idempotent functions).

Target Platform and Tools. We used the MSP-EXPFR5994 evaluation board [48], which includes 256kB FRAM and 4kB SRAM memory and can operate at up to 16MHz. We

	Bitcount (BC)		Cuckoo (CF)		Activity (AR)		DNN	
	Time (ms)	*Energy (µj)*	*Time (ms)*	*Energy (µj)*	*Time (ms)*	*Energy (µj)*	*Time (ms)*	*Energy (µj)*
Plain-Ram	24.73	41.54	36.81	63.30	822.78	1415.53	✗	✗
Plain-Fram	213.23	344.57	48.03	87.71	1053.75	2073.65	33624.60	59710
Alpaca	285.29	690.46	79.25	210.25	1897.90	5175.50	41787.88	77537
InK	497.19	1287.05	376.12	1016.49	3100.97	8707.40	46994.33	91961
TICS	482.38	1205.20	1229.30	2025.70	2667.16	7106.80	✗	✗
Immortal Threads (IT)	274.43	456.31	53.91	108.15	2503.45	4917.10	69215.54	147595

Table 4: Execution time and energy consumption of the benchmarks on **continuous power**.

	Immortal Threads			Alpaca			InK			TICS		
	Avg. Task Size (µs)	*Avg. Checkpoint Overhead (µs)*	*Tot. Invoked Checkpoint*	*Avg. Task Size (µs)*	*Avg. Task Trans. Overhead (µs)*	*Tot. Task Transition*	*Avg. Task Size (µs)*	*Avg. Task Trans. Overhead (µs)*	*Tot. Task Transition*	*Avg. Task Size (µs)*	*Avg. Checkpoint Overhead (µs)*	*Tot. Invoked Checkpoint*
BC	∼50.33	∼14.44	4236	280.97	127.56	709	169.69	537.78	709	2028.38	475.31	709
CF	∼31.97	∼9.52	1299	61.92	121.08	451	129.23	776.50	419	1110.58	454.72	518
AR	∼23.22	∼17.00	30223	829.87	124.55	2001	880.36	670.06	2007	245.14	411.14	1130
DNN	∼20.57	∼14.45	1865103	16742.40	570.05	2412	31765.28	1130.26	1486	✗	✗	✗

Table 5: Average execution time of a task, and task transition/checkpoint overhead.

used the 1 MHz frequency during the experiments on performance comparison (to be compatible with existing studies [31, 34, 54]. We used the GNU GCC v9.2.0.50 to compile our applications. To measure the time overhead and energy consumption, we used a logic analyzer and TI EnergyTrace software [49], respectively. We used the Powercast TX91501-3W [12] RF transmitter operating at 915 MHz center frequency to power wirelessly our evaluation board connected to the P2110-EVB [13] RF receiver. We used the 1mF and 50mF capacitors on P2110-EVB as energy storage to observe different power failure patterns. We also emulated power failures for the repeatability and replicability of comparative measurements. We generated a random soft reset triggered by an MCU timer with a uniformly distributed firing period in the interval of [5ms, 20ms] (as in Yildirim et al. [54]).

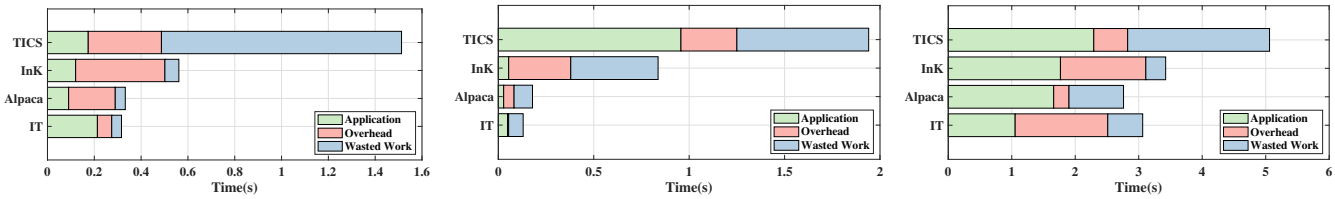
Evaluation Metrics. We considered *execution time* and *energy consumption* as the main metrics to evaluate the benchmarks. We also measured *wasted work* (which denotes computational progress lost due to power failures), *runtime overhead* introduced to progress the computation and keep memory consistent, and the memory requirements and code sizes of the benchmark implementations.

5.1 Evaluation Using BC, CF and AR

The InK and Alpaca implementations of the benchmarks have identical task boundaries. We placed TICS checkpoints aligned with task boundaries in the InK and Alpaca implementations for the sake of a fair comparison.

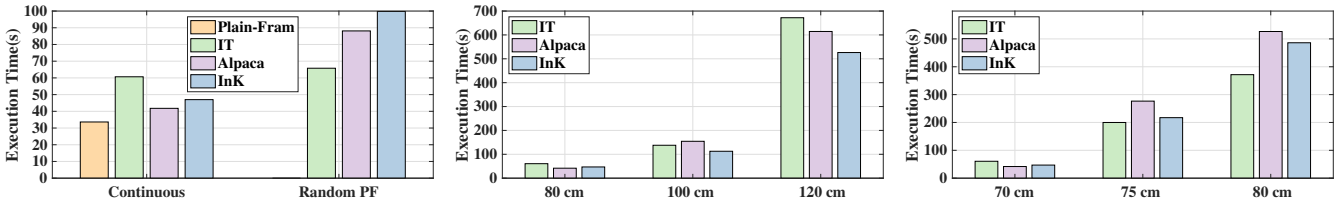
Continuous Power. Table 4 presents the continuously-powered execution time and energy consumption of the benchmarks. These benchmarks have different characteristics; for example, BC accesses memory more frequently to manipulate variables, and CF is more computationally dense. The differences in the time and energy overheads of the plain-Ram and plain-Fram versions show that intermittent computing, which requires frequent FRAM access, comes with significant overheads. Immortal Threads, InK, Alpaca, and TICS introduce additional overhead to Plain-Fram versions of the benchmarks to ensure forward progress and memory consistent. We observed that the performance of Immortal Threads is quite comparable to that of InK, TICS, and Alpaca during the continuous execution of the benchmarks. The reason is that InK and Alpaca need to perform bulk copy operations to commit the temporary buffers atomically during task transitions. Similarly, TICS needs to copy the stack and registers upon each checkpoint. Even though Immortal Threads maintains all variables in FRAM (which increases the time and energy overhead), almost-free checkpoints reduce the checkpointing cost, and just-in-time privatization eliminates block FRAM copy operations. Table 5 summarizes the average execution time of a task and the overhead of task transitions and checkpoints.

Intermittent Power. Figure 7 presents the wasted computational progress due to power failures and runtime overheads during intermittent execution with randomly generated power failures. The runtime overhead in InK and Alpaca is mainly due to the undo and redo logging operations performed by the tasks to recover computation upon power failures. Alpaca has a lower task transition overhead since it only double-buffers the task-shared variables with WAR dependencies and commits them upon task completion. Similarly, the overhead of TICS is due to the checkpoints and their restoration. TICS has more commit overhead since it checkpoints at the end of each task boundary, which requires a large bulk memory copy operation compared to task transitions in InK and Alpaca (see Table 5). During our experiments, Alpaca implementations of the benchmarks led to shorter task execution times and reduced wasted work since Alpaca introduced a lower runtime overhead compared to InK and TICS. In Immortal Threads, the runtime overhead is the total overhead of almost-free checkpoints, just-in-time privatization, and restoring compu-



(a) Bitcount (b) Cuckoo Filter (c) Activity Recognition

Figure 7: Total execution time, runtime overhead and wasted work with **controlled power failures**.



(a) Cont. power and controlled power failures. (b) RF-powered, **1mF** capacitor. (c) RF-powered, **50mF** capacitor.

Figure 8: DNN benchmark with **continuous power (Cont.)**, **controlled power failures** and **RF power** (at different distances).

tation. Conceptually, idempotent code blocks between two successive memory updates in Immortal Threads can be considered as a tiny task. These micro-continuations reduced wasted work and the total execution time significantly compared to existing runtimes.

5.2 Evaluation Using DNN Inference

Immortal Threads showed promising performance with relatively small benchmarks. To evaluate it under excessive memory access and computational load, we used the deep neural network (DNN) inference presented in [22]. This DNN model requires approximately 180kB FRAM to maintain the DNN weights and input matrix. The Alpaca DNN implementation in [22] employs *loop continuation* and has 18 tasks (2 tasks are for specific initialization operations). It is again worth mentioning that loop continuation relies on manually eliminated WAR dependencies and **violates** the task-based model. The implementation of TICS (from its public repository) could not support DNN inference, since its checkpoints lead to memory inconsistencies when the application accesses the higher regions of FRAM.InK requires DNN weights and the input matrix to be allocated in task-shared memory regions. However, InK double buffers all task-shared variables and commits them non-selectively at each task completion. Therefore, the implementation of DNN in InK is not feasible since it needs to commit a large amount of task-shared data at each task transition. However, by **violating** the InK model, we provided loop continuation support, allowed tasks to manipulate FRAM directly, and managed to implement DNN, which has 16 tasks (2 tasks specific to Alpaca are not required). Since our platform has only 4kB SRAM, we could not implement the Plain-RAM version of DNN.

Continuous Power. Due to the increased number of memory write operations, the overhead of Immortal Threads is more visible in this case (see Figure 8a). Immortal Threads introduced almost twice more overhead compared to Plain-Fram DNN (see Table 4). The main reason for performance degradation is committing each memory update atomically via the JIT privatization. INK and Alpaca performed better since they eliminated memory commit overheads by **violating** the task-based model via loop continuation. This violation allowed for larger tasks, which reduced the number of task transitions.

RF Powered. We used 1mF and 50mF capacitors as energy storage and three different distances from the RF power transmitter to observe different power failure patterns. The charging time of the capacitor increases with the distance between the receiver and the power transmitter. The charging time of the 50mF capacitor is longer than that of the 1mF capacitor. On the other hand, the 50mF capacitor provides a longer operation time. We observed significantly higher power failure rates with the 1mF capacitor. We conclude from Figure 8b and Figure 8c that Immortal Threads's performance becomes superior to the other runtimes as the power failure rate increases—it wastes less computational progress thanks to the micro-continuations.

Unviolated Task Model. We implemented DNN in Alpaca **without** the loop continuation to answer the question of what the DNN performance is without violating the task-based model. Our implementation, which we call the original Alpaca (*Alpaca (Org.)*), introduced an additional 11 tasks to the DNN implementation with loop continuation. Figure 9 presents execution time, runtime overhead, and wasted work during controlled power failures. We observed that Immortal Threads outperformed the original Alpaca significantly, i.e., led to a twice shorter execution time. Furthermore, even though

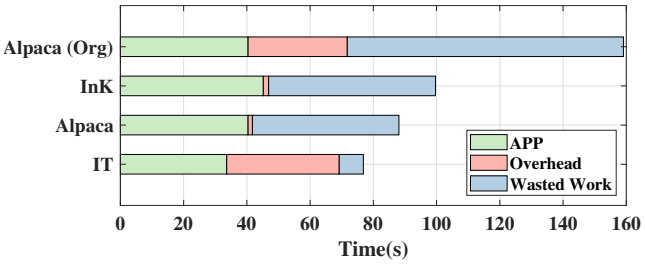


Figure 9: DNN execution time, runtime overhead and wasted work with **controlled power failures**.

Immortal Threads has more overhead compared to the InK and Alpaca implementations, it wasted significantly less work.

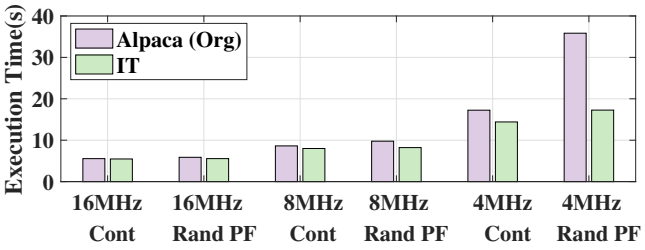


Figure 10: Performance of DNN at different frequencies.

Fram-CPU Bottleneck. As CPU speed increases, the FRAM access latency becomes more dominant in system performance. The FRAM in our platform can operate at a maximum speed of 8 MHz. We evaluated the original Alpaca and Immortal Threads DNN inference performances using clock frequencies of 4MHz, 8MHz and 16MHz with controlled power failures, as presented in Figure 10. We observed that as the clock frequency increases, the performances of both systems come closer to each other due to the latency of FRAM access, but Immortal Threads still performs better.

	Alpaca			InK			TICS		Immortal Th.	
	Tasks	*Trans.*	*Lines*	*Tasks*	*Trans.*	*Lines*	*Chkpts.*	*Lines*	*Chkpts.*	*Lines*
BC	11	24	251	10	26	326	10	238	82	188
CF	16	23	279	15	27	326	14	353	68	149
AR	12	20	330	11	20	449	8	411	123	309
DNN	18	48	2412	16	39	2214	✗	✗	276	1486

Table 6: Num. lines of code, num. of tasks and transitions (Alpaca and InK), num. of checkpoints (TICS and InK).

5.3 Cognitive Load, Code Size, and Memory Requirements

We define the *cognitive burden* of intermittent computing as the effort put to split code into idempotent sections, i.e., implementing tasks and task-based control flow in task-based systems and inserting checkpoints for checkpointing systems.

	Alpaca			InK			TICS			Immortal Th.		
	.text	*Ram*	*Fram*	*.text*	*Ram*	*Fram*	*.text*	*Ram*	*Fram*	*.text*	*Ram*	*Fram*
BC	2254	2	856	3356	0	4712	7160	4446	5572	10175	345	478
CF	3148	348	1070	4242	318	3000	11160	4655	6322	9831	370	498
AR	2258	0	784	3576	0	4474	11416	759	5430	11885	346	542
DNN	13898	224	192K	843	0	168K	✗	✗	✗	19394	356	149.5K

Table 7: Memory and Code Size requirements (in B).

We used the number of tasks (and checkpoints) and task-based control-flow declarations (in addition to the number of lines of code) shown in Table 6 as a metric to measure the burden. Thanks to the Immortal Threads compiler frontend, programmers write their programs without focusing on the details of the intermittent execution. The compiler frontend automatically wraps variable manipulations using the macros shown in 3, inserts checkpoints and creates idempotent code sections on the fly. Programmers use only the interfaces in Table 2, which are almost *identical* to the interfaces found in continuously powered event-driven systems [17, 32]. It is worth mentioning that Table 6 presents the number of lines after the Immortal Threads compiler pass (which has additional code inserted by the compiler). Even in this case, the number of lines in the implementations with Immortal Threads is almost half of that in the implementations with task-based models. As shown in Table 7, the code size of the application implemented in Immortal Threads library is larger than others. The main reason is that Immortal Threads library is implemented using C macros. It is worth mentioning that just-in-time privatization eliminates data versioning, reflected as considerably reduced data section requirements.

5.4 Summary of Evaluation Results

Our results showed that Immortal Threads has comparable runtime overhead to the existing runtimes. The runtime overhead and benefits of Immortal Threads depend on the application's memory access patterns and the frequency of the power failures. Compared to Immortal Threads, InK (violated task-based model) had approximately ten times more wasted computation and 1.5 times more execution time DNN inference under power failures. Besides, the original Alpaca (the unviolated task-based model) had approximately 17 times more wasted computational progress and 2.4 times more execution time. Therefore, we observed that during frequent power failures Immortal Threads reduced execution time and wasted work by up to 40% and 90%, respectively. We conclude that Immortal Threads brings pseudo-stackful multithreaded programming with acceptable overhead and no cognitive burden.

5.5 Greenhouse Monitoring Application

We proceed with greenhouse monitoring, which is a common application shown in intermittent computing studies (e.g., [31]), to demonstrate a time-constrained event-driven

scenario. To this end, we used a temperature sensor on the MSP430FR5994 MCU, a solar panel for energy harvesting, and an eZ430-RF2500 [47] board equipped with a CC2500 transceiver to transmit and receive data. MCU used a UART connection to send commands to eZ430-RF2500 for data transmission. We used the DS1302 [39] Real-Time Clock for time tracking despite power failures. As energy storage, we used the 50 mF supercapacitor of the P2110-EVB since it has a voltage regulator.Figure 11 shows our experimental setup.

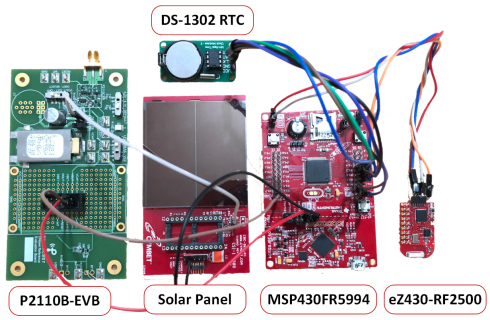


Figure 11: Greenhouse monitoring experimental setup.

GHM Implementation. A timer thread checks the RTC to signal timer events every 6 seconds. The sense thread blocks on the timer event to sense the temperature and store it in a buffer with a timestamp. When the number of samples reaches 10, the sense thread calculates the average and signals the send event. The send thread unblocks, checks the event timestamp (via `_EVENT_WAIT_EXP`), and sends data to eZ430-RF2500 if the event has not expired. Otherwise, it ignores the event.

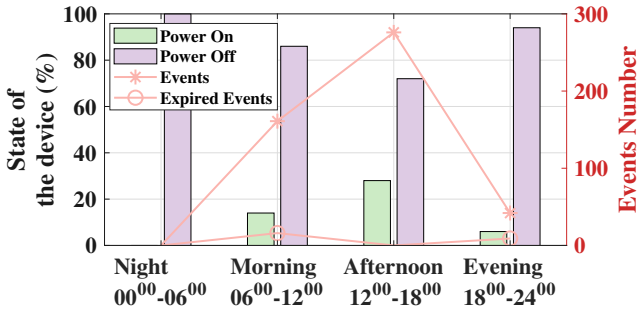


Figure 12: The number of expired events and the on/off time percentage of the device during different parts of the day.

Results. We placed our setup in an outdoor location on our campus for 24 hours. Figure 12 shows the results according to the parts of the day. To expose the effects of energy availability, we split the results into 6-hour timeframes. Since there was not enough energy at night, no events occurred. Due to power failures and charging times, 16 out of 177 events expired in the morning. The available environmental energy was high during the afternoon, and none of the 276 events expired since there were no power failures. Similarly, 9 out of 42 events have expired in the evening. We conclude that Immortal Threads successfully caught these expirations and stopped data processing to save precious harvested energy.

6 Discussion and Future Work

Programming models. Protothreads [18] is an abstraction designed for continuously-powered sensors. Its local continuation concept enables blocking threads, but such continuations can be saved only in the thread's entry function. InK task-threads provide a solution for intermittent event-driven applications, but they have mentioned drawbacks in this paper, e.g., the cognitive load of the task-based systems. Immortal Threads is an abstraction that provides micro continuation in intermittently powered systems, which is as lightweight as Protothreads's local continuation. In addition, they can be saved anywhere in the call stack of a thread, not only in the entry function. However, the current implementation of micro continuations achieves pseudo-stackfulness by employing `switch`-based constructs. An essential question for our future work is whether it is possible to control the compiler's usage of registers so that continuations can be composed of only the program counter and stack pointer while maintaining the remaining state in the (non-volatile) memory. Checkpointing at the statement boundary, combined with declaring variables as `volatile` or compiler fences, may provide a possible direction.

Peripheral operations support As previous works (e.g., [35]) point out, peripheral interaction should be atomic,which means no power failure can be allowed in between. In order to enable atomic execution of I/O handling operations, Immortal Threads compiler frontend can be extended to support a new compiler directive to mark atomic I/O functions, i.e., functions that should not contain checkpoints. The compiler frontend can add the necessary code that performs privatization of the parameters passed by address to such functions.

Checkpoint Optimization. Immortal Threads performs frequent checkpoints. Compiler analysis can be performed to merge and reduce unnecessary checkpoints. However, this may lead to more wasted work. In general, there is always a trade-off between checkpoint frequency (and the associated overhead) and the amount of waste work. Maeng et al. [36] proposes an adaptive approach: checkpoints are disabled at runtime when the system has still enough energy. However, this approach is effective only when *determining whether to checkpoint* has much less cost than *taking the checkpoint*, which does not apply to Immortal Threads, where checkpointing is merely an atomic write. One possible way to introduce adaptive checkpointing is to have multiple versions for each immortal function with a different checkpoint density. The runtime can then determine which version to call based on the energy availability. We leave this issue for future work.

7 Conclusions

Immortal Threads is the first intermittent computing runtime that enables pseudo-stackful multithreaded programming. Using Immortal Threads, programmers focus only on their multithreaded program logic that handles events instead of focusing on managing intermittent execution. Immortal Threads brings the missing event-driven primitives to intermittent computing, e.g., semaphores and event expiration handling. All these features come with an overhead comparable to the overhead of existing intermittent computing runtimes. We observed that, depending on the application and power failure frequency, Immortal Threads can even reduce execution time and wasted work by up to 40% and 90%, respectively.

Acknowledgments

We thank the anonymous reviewers of OSDI 2021, SOSP 2021, ASPLOS 2021 and OSDI 2022 for their valuable comments and feedback. We would like to thank Przemysław Pawełczak (TU Delft, The Netherlands) for encouraging us to send this work to OSDI 2022. We are also grateful to Rodrigo Bruno for shepherding our final draft.

Availability

We release Immortal Threads as an open source project for the community, whose artifacts can be downloaded from https://tinysystems.github.io/ImmortalThreads.

References

[1] Clang 7 libtooling. https://github.com/llvm-mirror/clang/blob/master/docs/LibTooling.rst, March 2019. Last accessed: May. 7, 2021.

[2] Atul Adya, Jon Howell, Marvin Theimer, William J Bolosky, and John R Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[3] Saad Ahmed, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, Naveed Anwar Bhatti, and Luca Mottola. Towards smaller checkpoints for better intermittent computing. In *17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 132–133. IEEE, 2018.

[4] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 70–81, 2019.

[5] Abu Bakar, Alexander G Ross, Kasim Sinan Yildirim, and Josiah Hester. Rehash: A flexible, developer focused, heuristic adaptation platform for intermittently powered computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(3):1–42, 2021.

[6] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980, 2016.

[7] Richard Barry. Freertos, a free open source rtos for small embedded real time systems. Available at: "https://www.freertos.org/", 2003.

[8] Naveed Anwar Bhatti and Luca Mottola. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. In *16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 209–220. IEEE, 2017.

[9] Michael Buettner, Benjamin Greenstein, and David Wetherall. Dewdrop: An Energy-Aware runtime for computational RFID. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[10] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 514–530, 2016.

[11] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 116–127, 2018.

[12] Powercast Corp. Powercast hardware. http://www.powercastco.com, 2014. Last accessed: Dec. 10, 2020.

[13] Powercast Corp. Powercast hardware. https://www.powercastco.com/wp-content/uploads/2016/11/p2110-evb1.pdf, 2015. Last accessed: Dec. 10, 2020.

[14] Eren Çürük, Kasim Sinan Yıldırım, Przemyslaw Pawelczak, and Josiah Hester. On the accuracy of network synchronization using persistent hourglass clocks. In

Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, pages 35–41, 2019.

[15] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–67, 2020.

[16] Jasper De Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(3):1–34, 2020.

[17] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*, pages 455–462. IEEE, 2004.

[18] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, 2006.

[19] Caglar Durmaz, Kasim Sinan Yildirim, and Geylani Kardas. Puremem: a structured programming model for transiently powered computers. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1544–1551, 2019.

[20] Kai Geissdoerfer, Raja Jurdak, and Brano Kusy. Long-term energy-neutral operation of solar energy-harvesting sensor nodes under time-varying utility. In *17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 156–157. IEEE, 2018.

[21] Kai Geissdoerfer and Marco Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 439–455. USENIX Association, April 2021.

[22] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213, 2019.

[23] Philipp Gutruf, Vaishnavi Krishnamurthi, Abraham Vázquez-Guardado, Zhaoqian Xie, Anthony Banks, Chun-Ju Su, Yeshou Xu, Chad R Haney, Emily A Waters, Irawati Kandela, et al. Fully implantable optoelectronic systems for battery-free, multimodal operation in neuroscience research. *Nature Electronics*, 1(12):652–660, 2018.

[24] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–13, 2017.

[25] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–13, 2017.

[26] Matthew Hicks. Clank: Architectural support for intermittent computation. *ACM SIGARCH Computer Architecture News*, 45(2):228–240, 2017.

[27] Neal Jackson, Joshua Adkins, and Prabal Dutta. Capacity over capacitance for reliable energy harvesting sensors. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, pages 193–204, 2019.

[28] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 330–335. IEEE, 2014.

[29] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Razvan Musaloiu-E., Philip Levis, Andreas Terzis, and Ramesh Govindan. TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2009.

[30] Vito Kortbeek, Abu Bakar, Stefany Cruz, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Bfree: Enabling battery-free sensor prototyping with python. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(4):1–39, 2020.

[31] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–99, 2020.

[32] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[33] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices*, 50(6):575–585, 2015.

[34] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[35] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1101–1116. Association for Computing Machinery.

[36] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 129–144, Carlsbad, CA, October 2018. USENIX Association.

[37] Kiwan Maeng and Brandon Lucia. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1005–1021, 2020.

[38] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks (TOSN)*, 16(1):1–24, 2020.

[39] Maxim Interated. Ds1302 trickle-charge timekeeping chip. https://datasheets.maximintegrated.com/en/ds/DS1302.pdf, 2019. Last accessed: September 2019.

[40] William P McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 167–180, 2006.

[41] William P. McCartney and Nigamanth Sridhar. Stackless Multi-Threading for Embedded Systems. *IEEE Transactions on Computers*, 64(10):2940–2952, October 2015. Conference Name: IEEE Transactions on Computers.

[42] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. Camaroptera: A batteryless long-range remote visual sensing system. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, pages 8–14, 2019.

[43] Benjamin Ransford and Brandon Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, pages 1–3, 2014.

[44] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on rfid-scale devices. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 159–170, 2011.

[45] Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1085–1100, 2019.

[46] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement*, 57(11):2608–2615, 2008.

[47] Texas Instruments. ez430-rf2500 development tool user's guide. https://www.ti.com/lit/ug/slau227f/slau227f.pdf, 2015. Last accessed: September 2015.

[48] Texas Instruments. Msp430fr58xx, msp430fr59xx, msp430fr68xx, and msp430fr69xx family user's guide. http://www.ti.com/lit/ug/slau367o/slau367o.pdf, 2019. Last accessed: September 2019.

[49] Texas Instruments. EnergyTrace Technology. https://www.ti.com/tool/energytrace, 2021.

[50] Texas Instruments, Inc. FRAM faqs. http://www.ti.com/lit/ml/slat151/slat151.pdf, 2014. Last accessed: 2018.

[51] Hoang Truong, Shuo Zhang, Ufuk Muncuk, Phuc Nguyen, Nam Bui, Anh Nguyen, Qin Lv, Kaushik Chowdhury, Thang Dinh, and Tam Vu. Capband: Battery-free successive capacitance sensing wristband for hand gesture recognition. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 54–67, 2018.

[52] Harrison Williams, Michael Moukarzel, and Matthew Hicks. Failure sentinels: ubiquitous just-in-time intermittent computation via low-cost hardware support

for voltage monitoring. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 665–678. IEEE, 2021.

[53] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 17–32, Savannah, GA, November 2016. USENIX Association.

[54] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 41–53, 2018.

[55] Eren Yildiz, Lijun Chen, and Kasim Sinan Yildirim. Immortal Threads GitHub Repository. https://tinysystems.github.io/ImmortalThreads/, 2022. Last accessed: June. 1, 2022.

Debugging the OmniTable Way

Andrew Quinn
UC Santa Cruz

Jason Flinn
Meta

Michael Cafarella
MIT

Baris Kasikci
University of Michigan

Abstract

Debugging is time-consuming, accounting for roughly 50% of a developer's time. To identify the cause of a failure, a developer usually tracks the state of their program as it executes on a failing input. Unfortunately, most debugging tools make it difficult for a developer to specify the program state that they wish to observe and computationally expensive to observe execution state. Moreover, existing work to improve our debugging tools often restrict the state that a developer can track by either exposing incomplete execution state or requiring manual instrumentation.

In this paper, we propose an `OmniTable`, an abstraction that captures all execution state as a large queryable data table. We build a query model around an `OmniTable` that supports SQL to simplify debugging without restricting the state that a developer can observe: we find that `OmniTable` debugging queries are more succinct than equivalent logic specified using existing tools. An `OmniTable` decouples debugging logic from the original execution, which SteamDrill, our prototype, uses to reduce the performance overhead of debugging. The system employs *lazy materialization*: it uses deterministic record/replay to store the execution associated with each `OmniTable` and resolves queries by inspecting replay executions. It employs a novel multi-replay strategy that partitions query resolution across multiple replays and a parallel resolution strategy that simultaneously observes state at multiple points-in-time. We find that SteamDrill queries are an order-of-magnitude faster than existing debugging tools.

1 Introduction

Developers spend the majority of their time debugging their software [26]. Usually, a developer debugs by iteratively executing their program and using debugging tools to observe its state during the failing execution.

A developer can often identify the root cause of a simple bug by making a few observations about their program's execution state. However, to identify the root cause of a complex bug, such as a atomicity violation or performance degradation, the developer will need to make sophisticated observations. Conceptually, we can model the logic for such sophisticated observations as a *debugging program*, designed to make sense of the failing program. For example, when debugging, a developer may observe all of the values to which a variable is assigned during an execution. Their debugging program consists of a set data structure to store the values, logic after each assignment in the failing execution that adds the assigned value to the set, and a print statement to print the set when the execution terminates.

Unfortunately, many debugging tools, such as `gdb`, "printf"-debugging, and binary instrumentation, support debugging programs that have both high programming complexity and high performance overhead. Such tools support procedural debugging programs that observe state as a failing program executes. Procedural debugging programs have considerable programming complexity, especially for sophisticated tasks that track execution state over time (§6.2). High complexity can lead to bugs [15, 44] that prevent a developer from understanding the failing program. Additionally, such debugging programs impose high performance overhead since sophisticated debugging programs observe *a lot* of execution state which existing tools extract within the same execution context as the failing program. Consequently, procedural debugging programs can slow execution by between a factor of 2–1000 (§6.3), which can preclude the use of sophisticated debugging programs [11].

Alas, prior debugging work retains, or even exacerbates, high programming complexity or high performance overhead to improve the other. Some proposals lower the performance overhead of debugging by employing parallelism (e.g., Speck [30], SledgeHammer [35]) or low-level optimizations (e.g., optimistic hybrid analysis [9], efficient path profiling [3]). At best, such techniques require redesigning debugging programs, at worst, they require novel research contributions to accelerate even a single task (e.g., taint tracking [4]).

High-level debugging tools decrease programming complexity by allowing a developer to observe and summarize execution state using a high-level programming model (e.g.,

Fay [12], G2 [17], EndoScope [7]). However, such tools retain high performance overhead since they perform a debugging program's observations while executing the original failing program. To curtail the effect of high performance overhead, high-level debugging tools restrict the execution state that a developer can observe, either explicitly (e.g., by minimizing the times when a debugging program can observe state [12, 14, 25]) or implicitly (e.g., by requiring extensive manual instrumentation to specify observable execution state [17, 24, 40]). Such systems are well suited for tasks that only need to observe partial execution state, such as distributed tracing [24] or identifying specific classes of bugs [25], but are less suited for debugging complex issues.

This paper proposes the `OmniTable` query model, a new debugging paradigm that reduces the programming complexity and performance overhead of debugging without restricting the execution state that a developer can observe. The new `OmniTable` abstraction empowers the model. An `OmniTable` reduces programming complexity by presenting all of an execution's state as a large queryable data object. An `OmniTable` reduces performance overhead by decoupling a debugging program's observations from the original programs' execution to enable automated optimizations of debugging programs.

The `OmniTable` query model enables debugging programs that can observe any execution state with low programming complexity by turning to relational logic. Concretely, an `OmniTable` is a database table representation of an execution that contains all architectural state (i.e., the value of all bytes of memory and all registers) before every instruction executed by the program. From a developer's perspective, an `OmniTable` is extracted as a program executes and can later be queried using an extended SQL language to observe the execution's state. The model bridges the gap between the architectural state in an `OmniTable` and common debugging abstractions (e.g., the functions executed, variables assigned, etc.) by re-purposing existing database primitives (e.g., high-level views) and creating new query operators (e.g., traversal functions).

Unfortunately, naively materializing an `OmniTable` would lead to considerable performance overhead, since it would require performing a core-dump before every instruction. Instead, our prototype, SteamDrill, employs *lazy materialization*. Lazy materialization defers the calculation of an `OmniTable`'s state until a developer queries it. Rather than extract an `OmniTable` in its entirety during execution, SteamDrill uses deterministic record and replay to store the execution associated with the `OmniTable`. Deterministic record and replay enables SteamDrill to compress and store years worth of `OmniTables` on a commodity hard drive [10]. When a developer issues a query over an `OmniTable`, SteamDrill generates instrumentation which it injects into a new replay of the execution associated with the `OmniTable` to produce the execution state needed for the query.

SteamDrill reduces performance overhead by decoupling a debugging query's execution from the original program execution. SteamDrill uses a query planning approach that decomposes a debugging query into independent stages. SteamDrill implements a novel multi-replay query resolution strategy that executes each stage in a separate replay so that it can use data that is computationally inexpensive to observe (e.g., data about functions in an `OmniTable`) to reduce the compute cost of observing data that is computationally expensive to observe (e.g., data about each instruction in an `OmniTable`). In essence, multi-replay resolution uses the decoupling between an `OmniTable` query and the original execution to repeatedly observe `OmniTable` state at increasing detail. SteamDrill also uses decoupling to observe execution state from multiple points-in-time in parallel using thousands of machines [35, 47].

We built a SteamDrill prototype on top of Spark [47] and Arnold [10]. We evaluate the prototype using 5 detailed case studies of bugs reported in popular open-source applications (Memcached, redis, Apache, and SQLite). We identified 14 debugging programs that a developer would use to identify the root cause of each bug, including ad-hoc programs (e.g., "How many control-flow instructions did my function issue?") and standard dynamic analyses (e.g., a memory leak detector). We implemented the debugging programs using `OmniTable` queries and `gdb`'s python bindings, which provide a high-level language over `gdb` features. We found that `OmniTable` queries require up to 11.67 times fewer lines (with a geometric mean of 3.74 times fewer lines), up to 5.73 times fewer terms (with a geometric mean of 1.70 times fewer terms), and up to 23.49 times less estimated development time (with a geometric mean of 2.75 times less estimated development time) than `gdb` scripts. We evaluated the performance of SteamDrill on 3 representative debugging queries and find that it is faster than `gdb` by a factor of 99 based upon geometric mean.

We make the following contributions:

- The `OmniTable` query model, which decouples a debugging program from a failing execution to reduce the programming complexity and performance overhead of debugging.
- SteamDrill, which optimizes `OmniTable` queries using query planning, cluster-scale parallelization, and a novel multi-replay query resolution approach.
- An evaluation of 5 case studies and 14 queries that shows that `OmniTable` queries are more succinct and SteamDrill has lower latency than state-of-the-art tools.

2 Motivation

In this section, we describe a motivational case study showing how the `OmniTable` query model simplifies debugging. In the case study, a developer uses an `OmniTable` to diagnose a performance problem in redis [36]. In the study, a developer deploys redis as an in-memory key-value LRU cache for a

Vars(ot)	
Column `Name`: Type	Description
`time`: Long	Time of instruction
`thread`: Long	Thread that executed instruction
`eip`: Long	Program counter of instruction
`name`: String	Variable name
`value`: Any	Assigned value

Funcs(ot)	
Column `Name`: Type	Description
`enterTime`: Long	Time of function entry
`exitTime`: Long	Time of function exit (or null)
`name`: String	Function name
`thread`: Long	Thread that executed function
`callStack`: String	Call stack of function
`args`: Map[String->Any]	Argument values
`rVal`: Any	Return value of function execution

Figure 1: The schema of the `Funcs(ot)` and `Vars(ot)` views. Each line in each table describes a column in the view.

slow back-end service. Over time, the average end-to-end latency of their deployment creeps upwards; the developer notices that the increase correlates with the back-end service processing a higher percentage of requests.

The bug is challenging to diagnose since the developer only starts with a high-level symptom and is unaware of which parts of the program are related to the error. To determine the root cause of the bug, the developer summarizes an execution's behavior over time. The `OmniTable` allows the developer to observe all of the execution state of the program without requiring instrumentation; the query model's support for SQL aggregations allows the developer to succinctly summarize large amounts of execution state. Moreover, the `OmniTable` enables repeated queries over the same buggy execution, instead of requiring the bug be reproduced for each query.

In contrast, summarizing execution state over time is challenging with existing tools (§7). To use a procedural debugging tool (e.g., `gdb`), the developer must identify numerous instrumentation points, track execution state over time in complex data-structures, and implement algorithms to group data and calculate statistics. Other debugging tools simplify execution summarization, but provide incomplete interfaces in that they do not expose the execution state (e.g., PTQL [25], Fay [12]) or do not support the operators (e.g., Pivot Tracing [24], Execution Mining [20]) required for this case study. Finally, instrumentation-based tools (e.g., G2 [17], Pivot Tracing [24]) require extensive manual instrumentation to perform the necessary observations.

The developer uses 5 `OmniTable` queries to identify the root cause of the performance degradation. Rather than query an `OmniTable` directly, the developer uses *derived views* to simplify their queries. A derived view labels execution state according to an abstraction of execution behavior, such as the functions, in-scope variables, or memory read by each instruction in an `OmniTable`. Below, we describe the derived views

time	eip	name	value
100	0x1000	"used"	1000
100	0x1004	"entry"	NULL
102	0x1004	"used"	1000

Figure 2: Example data from `Vars(ot)` .

enterTime	exitTime	name	args	rVal
100	200	"lookupKey"	{"key":"k1"}	100
100	200	"lookupKey"	{"key":"k2"}	100
100	200	"incrRefCount"	{"key":"k1"}	NULL

Figure 3: Example data from `Funcs(ot)` (omitting the `callStack` and `thread` columns). .

that the developer uses. Then, we describe the `OmniTable` queries that the developer uses and compare them to debugging programs expressed using existing debugging tools.

2.1 Views

The developer uses two derived views, `Vars(ot)` and `Funcs(ot)`, which can be calculated over an `OmniTable`: `ot`. Figure 1 shows their schemas.

Vars(ot). The `Vars` view contains the value of all in-scope variables at each instruction in an `OmniTable`. Each row identifies the value of a single in-scope variable at a single instruction, regardless of whether that instruction accesses the variable. Figure 2 shows a few rows of the `Vars(ot)` view for the `OmniTable` for the buggy execution of redis used during this case study. A developer references the `Vars` view of an `OmniTable`, `ot`, by specifying `Vars(ot)` in their query.

Funcs(ot). The `Funcs(ot)` view contains information about the functions executed in an `OmniTable`—each row contains state from either the entry to or exit from a function execution contained in the `OmniTable`, `ot`. For example, Figure 3 contains a few rows of the `Funcs(ot)` view for the `OmniTable` for the buggy execution of redis used during this case study. The `enterTime`, `callStack`, and `args` columns are extracted upon function entry; the `exitTime` and `rVal` columns are extracted upon function exit (and are `NULL` for functions that never return); and the `name` and `thread` are extracted at both entry and exit and joined to match the entry and exit of each function. The `rVal` and `args` columns use the polymorphic type, Any, to encode different function signatures. For example, a developer specifies `args["i"]` to get the value of the argument `i` passed to a function.

The `time`, `enterTime` and `exitTime` columns from `Vars(ot)` and `Funcs(ot)` expose an ordering of events contained in the views and provide a primary key that uniquely identifies each row in the views. Moreover, a developer can use the the `time`, `enterTime`, and `exitTime` columns to correlated data across the `Funcs(ot)` and `Vars(ot)` views for the `OmniTable`. For example, a developer can determine the value of each in-scope variable at the entry to each function by joining `Funcs(ot)` and `Vars(ot)` on `enterTime = time`; the second query uses this feature (§2.2.2).

```
1 Select enterTime, count(distinct args["key"]) Over(
2   Order By enterTime
3   Rows Between 10000 Preceding and Current Row)
4 From Functs(ot)
5 Where f.name="lookupKey"
```

Listing 1: The developer's first query.

2.2 Queries

Next, we describe how the developer diagnoses the cause of the performance bug. First, they use deterministic record and replay to capture the `OmniTable` for an execution of redis during which the issue occurs. Then, they construct and execute the following five `OmniTable` queries.

2.2.1 First Query

The developer's first query (Listing 1) uses a windowed aggregation to approximate the number of items that the deployment caches in redis (i.e., the working set size) during the performance degradation. The developer suspects that the working set size increases over time, which would lead to additional cache misses in redis. Each cache miss sends a request to the back-end service, so this hypothesis would explain the creeping latency of the deployment.

The developer begins by inspecting redis's source code to identify the function, `lookupKey`, that finds an item in the cache. For each `lookupKey` execution, the developer creates a window containing the preceding 10,000 executions of `lookupKey` and counts the number of distinct keys passed to each function call in each window. The `OmniTable` query model succinctly represents this logic using *SQL aggregates*. SQL aggregates calculate a mathematical operation (e.g., **`count`**, **`sum`**) over a group of rows. An aggregate can operate over a window of requests, in which each group is an ordered list of rows that match an **`Over`** clause, as is the case in this query. Alternatively, an aggregate can operate over a group of rows that match a **`Group By`** clause, as is the case in the developer's third query (§2.2.3).

In detail, the query uses the **`Over`** operator to create sliding windows, each of which contains 10,000 consecutive calls to `lookupKey`, by ordering `Funcs(ot)` by `enterTime` (Lines 2–3). The query filters non-`lookupKey` windows (Line 5). It counts the number of distinct keys passed to each function call in each window using the `key` argument (`args["key"]`) and the **`count`** and **`distinct`** operators.

Existing debugging tools either cannot support the query, impose high programming complexity, or impose high performance overhead. EndoScope [7] and Fay [12] could support the developer's query, but imposes a high overhead since they tightly couple debugging logic's execution with the original program execution. Most high-level debugging tools do not support windowed-aggregations and are either unable to compute the query (e.g., Pivot Tracing [24], Execution Mining [20]) or require a developer to write a custom operator to compute the query (e.g., G2 [17] requires expressing the window clause in terms of a vertex-based graph-traversal). Instrumentation-based debugging tools (e.g., Pivot Tracing [24]) would require the developer manually instrument the `lookupKey` function to produce the value of `key`.

```
1  from gdb import Breakpoint, parse_and_eval
2  from collections import deque, defaultdict
3  class bp(Breakpoint):
4      keys=deque()
5      indexed=defaultdict(int)
6      def stop(self):
7          keys.append(parse_and_eval("key"))
8          indexed[keys[-1]] += 1
9          if len(keys) > 10000:
10             indexed[keys[0]] -= 1
11             if indexed[keys[0]] == 0:
12                 del indexed[keys[0]]
13             keys.popleft()
14         print (len(indexed))
15         return False
16 bp("lookupKey")
```

Listing 2: The developer's first query written for gdb's Python bindings.

An equivalent procedural debugging program is complex. The debugging program must navigate the performance-complexity tradeoff—creating a program with high overhead is straightforward, but creating one with low overhead requires complex logic to ensure consistency of two data structures. A mistake can lead to a misdiagnosis of the bug—our first version of the debugging program included such a mistake.

Listing 2 shows an implementation for `gdb`'s Python bindings, which provide a Python interface for `gdb` features such as breakpoints and backtraces. The developer creates a custom `Breakpoint` class, `bp` (Lines 6–15); by creating a `bp` with the argument `"lookupKey"` (Line16), the developer instructs the `gdb` framework to call the developer-supplied `stop` function at each call to `lookupKey`. The developer tracks the sliding window of 10,000 requests by storing the value of the `key` argument into `keys`, a `queue`, and removing the first element if there are more than 10,000 elements in `keys` (Lines 7, 9, and 13). The developer could recompute the unique values in `keys` in `stop`, but that would add significant performance overhead since `lookupKey` is executed frequently. Instead, the developer uses a dictionary object, `indexed`, to track the number of times each `key` value appears in the `keys` window (Lines 5, 8, and 10–12). This logic is subtle and challenging to get right—for example, we initially used a `set` to track the unique elements in `keys` instead of using a dictionary to track the number of times each element appears in `keys`. Our buggy implementation erroneously removes elements from `indexed` and produces misleading results.

```
1  Select v.time, v.value
2    From Vars(ot) as v Join Funcs(ot) as f
3   Where v.name = "used" And f.name = "lookupKey"
4     And f.enterTime = v.time
```

Listing 3: The developer's second query.

```
1  Define DefinedMemory(ot) as:
2  Select m.rVal as pointer, m.exitTime as start,
3         m.callStack as allocSite, f.enterTime as end
4         m.arg["size"] as size
5  From Funcs(ot) as m Left NextJoin Functs(ot) as f
6    On m.exitTime, f.enterTime,
7    m.name="malloc" And m.exitTime<=f.enterTime
8   And f.name="free" And m.rVal=f.arg["ptr"]
9
10 Select start, allocSite, sum(size)
11   Over(Partition By allocSite Order By start)
12 From DefinedMemory(ot)
13 Where end=NULL
```

Listing 4: The definition of the `DefinedMemory(ot)` view (Lines 1–8) and the developer's third query (Lines 10–13).

2.2.2 Second Query

Surprisingly, the working set size of the cache is fairly constant throughout the execution. Consequently, poor cache performance may arise from poor eviction decisions for the workload or from a decrease in the number of items in the cache over time. The developer's second query determines the number of items in the cache over time by checking the number of items in the cache before each execution of the `lookupKey` function (Listing 3). redis stores the size of the cache in a global variable, `used`; the query uses the `Vars(ot)` view to access the value of the variable (Lines 2–3). It prevents the query from producing extremely large amounts of data by using a `Join` to limit the rows to only those when the execution enters `lookupKey` (Lines 2–4).

Unlike the `OmniTable` query model, many debugging tools do not expose the value of variables at arbitrary points-in-time and cannot support the developer's second query (e.g., Fay [12], PTQL [14]). Endoscope [7] could support the query, but only exposes variable values when they are assigned and requires the query identify the most recent preceding assignment of `used` for each call to `lookupKey`. Instrumentation-based tools (e.g., Pivot Tracing [24], G2 [17]) and procedural tools (e.g., `gdb`) require instrumenting the `lookupKey` function to produce the value of `used`.

2.2.3 Third Query

The second query's output shows that the number of items in the cache decreases throughout the execution. Since the redis configuration specifies a total memory size for the cache and the deployment uses constant sized items, a declining number of items in the cache implies that there is a memory leak. Unfortunately, redis does not clean up memory on shutdown, so existing leak detection tools (e.g., memcheck [29] and AddressSanitizer [39]) report nearly all memory allocations as leaks.

```
1  Select dm.pointer, Count(*)
2  From Funcs(ot) as r Join DefinedMemory(ot) as dm
3  Where dm.allocSite=leakSite And dm.exit=NULL
4    And r.name="decrRefCount" Or r.name="incrRefCount"
5    And dm.pointer=r.arg["obj"]
6  Group By dm.pointer, r.name
```

Listing 5: The developer's fourth query.

The developer's third query uses an alternative approach: it tracks the number of leaked bytes by each allocation site (defined as the call stack of the allocation) over time. Allocation sites that produce bug-inducing leaks will have a gradual increase of leaked bytes throughout the execution. The developer's query observes three separate types of execution events with different happens-before relationships, which is greatly simplified by the `OmniTable` query model.

The developer first creates a view, `DefinedMemory(ot)` that contains the window of time during which each memory object is defined, i.e., allocated and not freed (Listing 4). The view joins each call to malloc with the subsequent call to free whose pointer argument, `ptr`, is equal to the return value from malloc (Lines 5–8). Since a pointer could be reallocated by malloc after being freed, the query only matches calls to free that occur after the call to malloc (`m.exitTime<f.enterTime` at Line 7). Additionally, it only matches each malloc with the next matching call to free, as ordered by `exitTime` and `enterTime`, respectively, by using **`NextJoin`**, a new operator provided by the `OmniTable` query model (Lines 5–6). The developer uses **`Left NextJoin`**, which produces output from the left relation even if there is no matching row in the right relation, so that memory which is never freed (i.e., leaked) has a `NULL` value for the `end` column.

The third query tracks the amount of data leaked by each allocation site over time. For each leaking allocation (Lines 12–13), the developer constructs a window containing all preceding leaking allocations from the same allocation site by using the **`Over`** operator (Line 11). They sum the number of bytes leaked within each window (Line 10).

Like with the previous queries, existing debugging tools either cannot support the third query, impose high programming complexity, or impose high performance overhead.

2.2.4 Fourth Query

The output of the third query identifies a single leaking allocation site, `leakSite`. redis uses reference counters to manage allocations from `leakSite`. Each counter tracks the number of live references to each object; redis should garbage collect the object when the count reaches 0. So, the developer suspects a problem in the reference counting and writes a query to count the updates to the reference counters of leak-

ing objects (Listing 5). They identify leaked objects that were allocated at `leakSite` (Lines 2–3) and match each leaked object with corresponding executions of `decrRefCount` and `incrRefCount`, the functions that modify reference counts (Lines 2, 4, and 5). The developer groups the rows by object and function name (Line 6) and determines the number of calls to increment and decrement the counter (Line 1).

Like the previous queries, existing debugging tools either cannot support the developer's fourth query, impose high programming complexity, or impose high performance overhead.

2.2.5 Fifth Query

The fourth query's output shows that the execution calls `incrRefCount` and `derRefCount` the same number of times on the leaked objects, indicating a problem in the implementation of `incrRefCount` or `decrRefCount`. The developer chooses a few candidate objects and determines the call stack of the calls to `incrRefCount` and `decrRefCount` for these objects. The final query[1] shows that the leaked object's reference counts are decremented by a lazy deallocation thread and by a logging thread and points to the root cause of the bug, a race condition in `decrRefCount`. In the fix for the original bug report, the developer redesigning the logging thread to copy objects instead of sharing them.

3 The `OmniTable` Query Model

We outline the features of the `OmniTable` query model that enable a developer to succinctly reason about the entire history of execution state. The central abstraction is an `OmniTable`, a database table containing all user-level architectural state of an execution immediately before every instruction in the execution. Concretely, an `OmniTable` contains a column for every byte of architectural state and a row immediately before each instruction. The model supports debugging queries over an `OmniTable` expressed using SQL-style **`Select`**...**`From`**...**`Where`** queries.

Alas, an `OmniTable` alone offers an inadequate debugging interface, since a developer would need to reference execution state in architectural terms. For example, a developer would need to determine the exact memory location of each variable whose value they wish to observe. So, the `OmniTable` model adopts and extends database concepts to enable debugging abstractions. It uses `Generators`, user-defined-table-functions that allow queries to reference non-execution state (e.g., debugging symbols). It adds new operators for debugging, such as traversal functions and new `Join` variants. Finally, the model uses derived views to label an `OmniTable`'s state according to familiar debugging abstractions such as the functions executed in an `OmniTable` or the variables in scope at each instruction in an `OmniTable`. A single row in a high-level view can expose execution state from multiple points-in-time during the execution (e.g., `Funcs(ot)`). Below, we elaborate on the model's components.

[1] Omitted for brevity, this query is a self-join of the `Funcs(ot)` view

3.1 Relations

The `OmniTable` query model supports two relational base tables, `OmniTables` and `Generators`. It supports columns with primitive types (e.g., Long, String), Structs, Maps, Arrays, and Any, a polymorphic type.

OmniTable. An `OmniTable` is a database table that includes all architectural execution state immediately before each instruction in the execution; Figure 4 shows an example. Before each instruction, the `OmniTable` contains the current `thread`, the value of all registers and memory addresses, and the top of the `stack` of the thread. To dereference an address, `addr`, a query specifies `Memory[addr]`. Additionally, each row includes a monotonically increasing logical `time`, which provides a total ordering of events in the `OmniTable` and uniquely identifies each row. In a multi-threaded program, the `time` field is a total ordering that is consistent with the partial ordering of the original execution. Together, the `thread` and `time` columns enable a developer to reason about concurrency.

Generators. `Generators` allow developers to bridge the semantic gap between traditional programming abstractions (e.g., functions, lines of code) and an `OmniTable`'s architectural state by referencing non-execution state (e.g., debugging symbols). For example, `Defs` identifies the functions defined in a binary; the following produces all such definitions for an executable, "a.out": **`Select`** `*` **`From`** `Defs("a.out")`. `Generator` input can depend on query data. For example, `Binaries` is useful for bootstrapping queries; it uses the deterministic record/replay log to identify the binaries mapped into the address space of an `OmniTable`. The following determines all functions defined in all binaries that are loaded in `ot`, an `OmniTable`, which we use to define the `FuncDefs` view: **`Select`** `*` **`From`** `Defs(`**`Select`** `*` **`From`** `Binaries(ot))`. Developers create `Generators` by writing a program that produces relational output; we have built a `Generator` that determines all variables defined in in all binaries mapped into an `OmniTable`, and one that creates stored procedures that produce the memory read and written by each instruction in an `OmniTable`.

3.2 Relational Operators

The model supports **`join`**, **`group by`**, **`order by`**, and **`pivot`**. It also introduces three `Join` variants for debugging.

StackJoin. SQL is unable to model a function stack, which would prevent the `OmniTable` query model from expressing critical debugging abstractions, such as `Funcs(ot)`, which is used in all of the queries in the redis case study (§2). Prior high-level debugging tools either remove support for such se-

Metadata				Registers				Memory			
time	**thread**	**stackTop**	**...**	**eip**	**eax**	**ebx**	**...**	**0x0**	**0x1**	**...**	**0xffffffff**
1	100	0x2000	...	0x1000	1	1	...	1	1	...	1
2	100	0x2000	...	0x1004	1	1	...	1	1	...	1
...											
1000	100	0x2000	...	0x1064	1	1	...	1	1	...	1

Figure 4: An `OmniTable` for a short execution.

```
1 Select *
2 From fenter(ot) as e StackJoin freturn(ot) as r
3 On e.time, r.time, e.thread=r.thread AND e.name=r.name
```

Listing 6: An Example StackJoin.

mantics (e.g., PTQL [14]) or require manual instrumentation (e.g., G2 [17]). Instead, the `OmniTable` model creates a new operator, **`StackJoin`**.

As an example, suppose that `fenter(ot)` is a view that contains a row for each function entry in `ot`, an `OmniTable`, with columns `name`, `thread`, and `time` for the name of the function, thread that entered the function, and time of entry; and that `freturn(ot)` is a view conatining a row for each function return in `ot`, an `OmniTable`, with columns `name`, `thread`, and `time` for the name of the function, thread that returned from the function, and time that the function returns. Listing 6 shows a **`StackJoin`** that matches each function entry with its function return. **`StackJoin`** partitions `fenter(ot)` and `freturn(ot)` into groups that match on `thread` and `name`. For each group, the operator orders the rows `fenter(ot)` by `time` and orders the rows from `freturn(ot)` by `time`. Repeatedly, the operator pushes all rows from `fenter(ot)` onto a stack until it finds a row that occurs after the next row in `freturn(ot)`; it then produces a row by joining the last row added to the stack and the next row in `freturn(ot)`.

OrderedJoins. When debugging, developers often reason about the next, or previous, event that satisfies some condition. For example, in the `DefinedMemory(ot)` view, the developer matched each call to `malloc` with the next call to `free` on the same pointer (§2.2.3). SQL requires inconvenient subqueries for this reasoning, so, the `OmniTable` query model adds two new ordered join operators. The `NextJoin` operator determines the next matching row across two relations and can be used to determine the next function executed by a thread or the next access to a shared variable: **`NextJoin on`** `ord1, ord2, equals` joins each row in the left relation, ordered by `ord1`, with the next row from the right relation, ordered by `ord2`, where `equals` is true. The `PrevJoin` operator does the opposite.

3.3 Column Operators

The `OmniTable` query model supports many column operators, including arithmetic and conditional operators, field expressions (`a.b`), subscript expressions (`a[b]`), traversal functions, stored procedures, and standard aggregations (e.g., **`Count`**, **`Max`**, **`Min`**, etc.) over groups and windows. The model also supports pointer dereferences by converting them into expressions over the `Memory` column (e.g., `a->b` becomes `Memory[a].b`). We elaborate on traversal functions and stored procedures.

Traversal Functions. SQL makes it difficult to traverse the elements in a data structure since it does not support unbounded traversals. So, the `OmniTable` query model builds new primitives for these operations. Given a pointer-typed `column` and a `field` within the pointed-to type, the `traverse(column, field)` expression produces a row of output for each element in the transitive closure of the structure by starting at `column` and following `field` pointers until the value is `NULL`. For example, `traverse(node, "next")` traverses the `next` pointer of all elements in a structure, starting at `node`.

Stored Procedures. Debugging logic often varies by execution context (e.g., the memory location of function arguments varies by function). Stored procedures [43] store relational logic in a table and allow a query to decide query logic during query resolution. Developers call stored procedures in their `OmniTable` queries with function syntax; for example, a developer could specify `Var_Loc(esp)` to use `Var_Loc`, a stored procedure that calculates the memory location of a variable given the value of the stack pointer.

3.4 Derived Views

The `OmniTable` query model allows developers to construct derived views for labeling execution state. The **Define** operator in Listing 4 shows how a developer constructs `DefinedMemory(ot)`. Our implementation provides three high-level views, `Funcs(ot)` (§2.1), `Vars(ot)` (§2.1), and `Insts(ot)`, a view that encodes information about each instruction in an `OmniTable`.

4 Design

In this section, we describe the design of SteamDrill, our system that supports the `OmniTable` query model. From a developer's perspective, SteamDrill computes queries over `OmniTables` that are extracted during execution and stored in a database. However, materializing an entire `OmniTable` is infeasible due to high storage and compute costs: an `OmniTable`'s size is equal to the addressable memory size

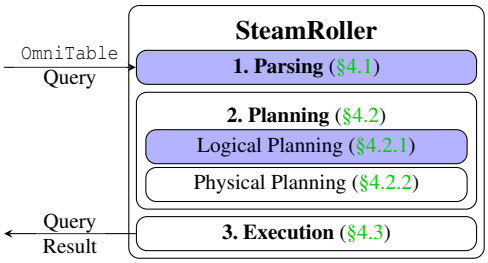


Figure 5: SteamDrill steps for query resolution. Blue steps re-use or customize existing approaches, and white steps are new designs.

```
1 Select eip, read, write
2 From Insts(ot) as i Left Join DefinedMemory(ot) as dm
3    On (i.read=dm.pointer Or i.write=dm.pointer)
4    And i.time>=dm.start And i.time<dm.end
5 Where dm.start=NULL
```

Listing 7: A simplified undefined use query.

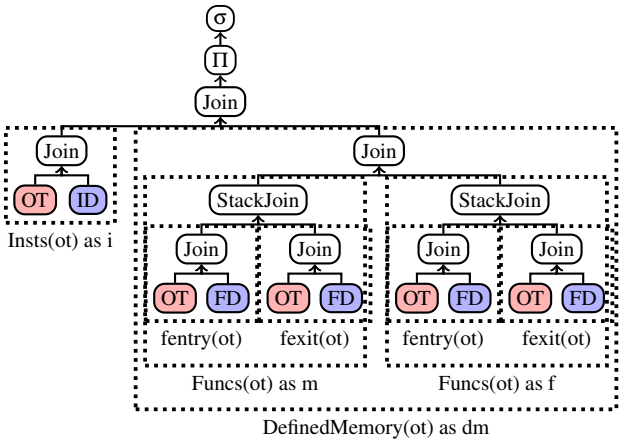


Figure 6: The relational tree for Listing 7. `OmniTables` are red ovals labeled with OT. `Generators` are blue ovals; `InstructionDefs` are ID nodes and `FuncDefs` are FD nodes. Relational operators are white rectangles; **`where`**, **`join`**, and **`select`** clauses are σ, Join, and Π nodes. The logic for each derived view is encapsulated in a dotted rectangle.

times the number of executed instructions and reaches petabytes, or even exabytes, for mere seconds of execution.

SteamDrill introduces lazy materialization as a solution. Rather than materializing an `OmniTable` during execution, SteamDrill uses deterministic record and replay [10] to capture a log of non-deterministic inputs to the execution. The system uses the log to generate `OmniTable` state on-demand by instrumenting and re-executing the original execution as necessary to resolve debugging queries. Delaying `OmniTable` materialization allows SteamDrill to filter `OmniTable` data *before* extracting state instead of afterwards.

In the rest of this section we describe how SteamDrill resolves debugging queries. Listing 7 presents a simplified use-after-free query as a running example. The query uses the `Insts(ot)` view to identify the memory read and written by each instruction in an `OmniTable`. It joins `Insts(ot)` with the `DefinedMemory(ot)` view from the third redis query, to match each memory access with the region of time during which its pointer was defined (Listing 4). The query uses a **`Left Join`** and identifies rows where `start` is `NULL` to identify the instructions that operate on undefined memory.

SteamDrill's design mirrors that of typical database management systems [2] (Figure 5). First, SteamDrill uses conventional SQL parsing to decompose a query into a tree of relational operators (internal nodes) over data tables (leaves) (§4.1). The tree contains a separate leaf node for each `OmniTable` referenced in the query. The relational operators that consume data from each `OmniTable` in the tree identify the execution state that the query needs from that `OmniTable`. During execution, SteamDrill uses these operators to limit the materialization of each `OmniTable` in the query by generating instrumentation that it injects into replay executions (§4.3).

The order of `OmniTable` materialization has a large impact on the amount of materialized data and the latency of query resolution. Delaying the materialization of an otherwise computationally expensive-to-materialize `OmniTable` often allows SteamDrill to filter the computationally expensive materialization using data from a less computationally expensive-to-materialize `OmniTable`. For example, to reduce the latency of the use-after-free query, the system first materializes the `OmniTable` state needed for the `DefinedMemory` view and uses the materialized data to filter the materialization of the `Insts(ot)` view.

Accordingly, SteamDrill uses multi-replay resolution—it splits `OmniTable` materialization across multiple replay executions. It uses query planning to determine the `OmniTable` materialization order and assigns `OmniTable` materialization to replay executions(§4.2). SteamDrill implements `OmniTable`-specific optimization strategies to decide the join order and join algorithm for each join in the tree (§4.2.1) which it uses to assign each operator to a stage. The system uses a single replay execution to materializes all `OmniTables` in the same stage. This approach minimizes the number of replay executions (since each replay execution adds additional overhead) and enables SteamDrill to limit `OmniTable` materialization.

4.1 Parsing

First, SteamDrill converts the query into a tree of relational operators over data tables using a standard SQL parser [2]; the tree encodes the logic required to resolve the query in terms of easy-to-optimize relational operations. The tree encodes each relational operator in the query as an internal node (i.e., a projection (Π), selection (σ), or join operator) and each `OmniTable` and each `Generator` in the query as a separate

leaf node. SteamDrill recursively decomposes each view into the relational logic that generates them until the tree is comprised entirely of relational logic and base tables (§3.1). A directed edge from node n_1 to node n_2 in the tree identifies that the operator n_2 consumes the output of n_1.

Figure 6 is the relational tree produced by SteamDrill for Listing 7. SteamDrill contains the internal logic of the `Insts(ot)` and `DefinedMemory(ot)` views (shown as dotted rectangles). The tree contains the logic of the `Insts(ot)` view: a **`Join`** between an `OmniTable` and `InstructionDefs`, a `Generator` containing metadata about the instructions defined in binaries used by an `OmniTable`.

The tree includes the internal logic of `DefinedMemory(ot)` and, recursively, all of the derived views comprising `DefinedMemory(ot)`. The tree contains the `DefinedMemory(ot)` logic (Listing 4): a **`Join`** between two `Funcs(ot)` views, one for executions of malloc (`Funcs(ot)` as m) and one for executions of free (`Funcs(ot)` as f). The tree contains the logic of each `Funcs(ot)` view (§3.2 and Listing 6): a **`StackJoin`** that combines `fentry(ot)` and `fexit(ot)`, relations over the entry and exit to each function in the `OmniTable`. Finally, the tree contains the logic for each `fentry(ot)` and `fexit(ot)`: a **`Join`** between an `OmniTable` and `FuncDefs`. Note, the tree includes the `fentry(ot)` of malloc and `fexit(ot)` of free even though the query does not use their output; during planning, SteamDrill determines that the query does not use the views and prunes them.

4.2 Planning

SteamDrill performs two tasks during planning. During logical planning, the system optimizes the relational tree using standard optimizations (e.g., predicate push-down) and determines the join order and join algorithm for each join in the tree using `OmniTable`-specific strategies (§4.2.1). The most crucial task in logical planning is determining the join order and algorithms for the query, since the join order and algorithms imply the partial order in which SteamDrill will materialize the `OmniTable` nodes contained in the query. SteamDrill supports two join algorithms: merge joins, which operate over two fully realized relations, and block-nested-loop joins (loop joins), which first calculate the left relation and use the left relation's output to limit right relation materialization.

During physical planning, SteamDrill produces a staged execution plan, which uses the join order and algorithms assigned during logical planning to assign each operator in the tree to a stage. In particular, physical planning assigns the children of merge joins to the same stage (so SteamDrill materializes them using the same replay) and assigns the right child of a loop join to the stage after the loop join's left child (so SteamDrill materializes them using different replays).

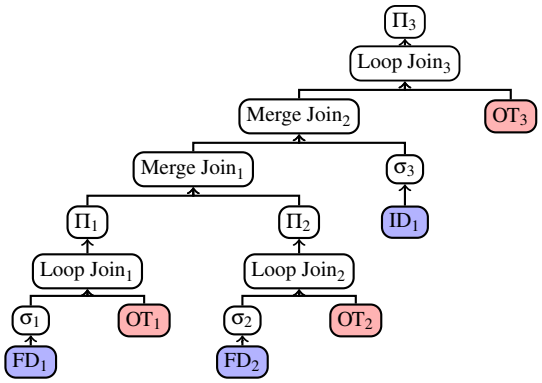


Figure 7: The relational tree for Figure 6 after logical planning.

4.2.1 Logical Planning

Traditional techniques for deciding join order and algorithms perform poorly on `OmniTable` queries for three reasons: First, similar subtrees in a relational tree have vastly different computational costs to materialize (e.g., the **`join`** subtree in the `Funcs(ot)` subtree is similar but much less computationally expensive than the **`join`** subtree in the `Insts(ot)` subtree). Second, the materialization cost of an `OmniTable` often depends on unpredictable properties of the underlying execution (e.g., it is difficult to predict the execution frequency of a particular function). Third, the enormous compute cost of materializing an `OmniTable` invalidates conventional rules.

Consequently, SteamDrill turns to a rule-based planner [1] that enables developers to encode semantic information that would be difficult or impossible for SteamDrill to deduce on its own. Each rule specifies regular-expression-like rules that pattern match subtrees of the relational tree and produce modified operators [1]. The join order and algorithm rules produce a left-deep join structure in which `OmniTable` nodes are isolated on the right-hand side of a join node (Figure 7), since these structures allow SteamDrill to perform as much filtering as possible when extracting data from an `OmniTable`. When queries join relations with different expected materialization compute costs, the rules place the less expensive relations on the left side, the more expensive relation on the right side and employ a loop join. When joining relations with the same expected materialization compute costs (e.g., two instances of `Funcs(ot)`), the rules use a merge join. Heuristically, SteamDrill expects that `Funcs(ot)` relations are less computationally expensive to materialize than `Vars(ot)` relations, which are less computationally expensive than `Insts(ot)` relations, which are less computationally expensive than `OmniTables`. Rules also encode traditional database optimizations.

Executing the relational tree in Figure 6 without logical planning would have high latency; SteamDrill would materialize the `OmniTable` five separate times! In contrast, SteamDrill's logical plan (Figure 7) uses multi-replay resolution to

observe only the exit from malloc, entry to free, and load/store instructions. Moreover, the plan reduces latency by only producing data for load/stores to undefined memory as they are observed, rather than producing data for all loads/stores and performing a join to determine undefined uses afterwards.

First, SteamDrill uses traditional database optimizations (e.g., operator push-down) to push operators towards leaf nodes to (1) produce `FuncDefs` data for only malloc and free (σ_1 and σ_2), (2) produce `InstructionDefs` data only for loads and stores (σ_3) and (3) eliminate the fentry(ot) for malloc and fexit(ot) for free. The system uses loop joins for Loop Join_1 and Loop Join_2, which materialize σ_1 and σ_2 before OT_1 and OT_1 to limit `OmniTable` state to the exit of malloc and entry to free in OT_1 and OT_2, respectively. The system joins them using a Merge Join (Merge Join_1) to limit the number of replay executions that it uses. OT_3, created for the `Insts(ot)` view, is computationally expensive to materialize, so SteamDrill defers its materialization. SteamDrill uses a Merge Join (Merge Join_2) to join σ_3 and Merge Join_1, which requires a Cartesian product and violates the traditional rule that such approaches be avoided. Materializing Merge Join_2 and using a loop join (Loop Join_3) to join it with OT_3 allows SteamDrill to identify only the loads/stores to undefined memory (i.e., loads/stores that read/write an address at a time when it is not contained in `DefinedMemory(ot)`) as they are performed by the execution rather than in an expensive join afterwards. In some queries, using a loop join like Loop Join_3 enables SteamDrill to elide inspection of some instructions altogether (e.g., Listing 8).

4.2.2 Physical Planning

Next, SteamDrill converts the optimized relational tree into a staged execution plan by assigning each operator from the tree into a *stage*. Each stage corresponds to a new replay execution (§4.3). SteamDrill assigns operators to stages that follow the partial order of `OmniTable` materialization that is implied by the join order and algorithm, but uses as few stages as possible, since each stage will require the additional latency and overhead of a replay execution.

SteamDrill performs a depth-first traversal of the tree starting at the root node and maintains an integer id for the current stage, starting at 1. The system assigns leaf nodes (`OmniTable`, `Generators`) to the current stage and unary nodes (i.e., all non-join operators) to their child's stage. The system assigns merge join operators to the largest stage among the join's children. For loop join operators, SteamDrill first assigns stages to operators in the left (inexpensive) child, adds one to the current stage, assigns the loop join to the new stage and traverses the right (expensive) child.

Figure 8 shows the staged execution plan for Listing 7. SteamDrill assigns FD_1, σ_1, FD_2, σ_2, ID_1, and σ_3 to the first stage. It assigns OT_1 and OT_2 to the second stage since Loop Join_1 and Loop Join_2 indicate that OT_1 and OT_2 should

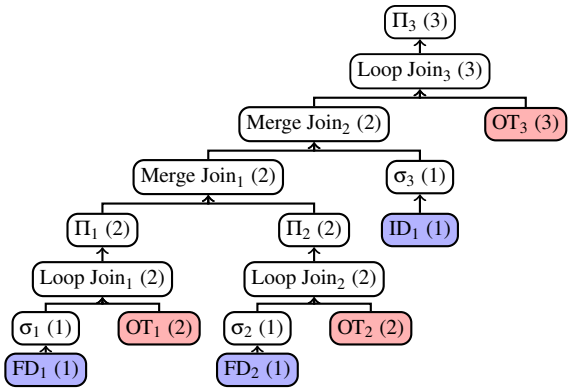


Figure 8: The staged execution plan for Figure 7. The stage of each node is shown in parentheses in the node.

be materialized after σ_1 and σ_2, respectively. SteamDrill also assigns Π_1, Π_2, Merge Join_1, and Merge Join_2 to the second stage since they inherit the largest stage of their children. The system assigns Loop Join_3, OT_3, and Π_3 to the third stage to follow the order required for Loop Join_3.

4.3 Execution

Finally, SteamDrill executes the staged execution plan. For each stage, the system generates instrumentation to materialize the state needed from each `OmniTable`, materializes each `OmniTable`, and calculates each operator in the stage.

4.3.1 Instrumentation Generation

SteamDrill generates instrumentation that it will inject into a replay execution for the `OmniTables` in a stage by determining instrumentation operators for each `OmniTable` node in the stage. For each `OmniTable` node, the system gathers all stateless operators (e.g., projections (Π) and selections(σ)) that only consume data from (1) the `OmniTable` node, (2) nodes resolved in previous stages, or (3) other nodes satisfying (1) and (2). For example, the instrumentation operators for OT_1 in Figure 8 includes Π_1 and Loop Join_1. Selecting stateless operations ensures that the resulting instrumentation will be parallelizable during materialization.

Then, SteamDrill creates a cursor object for each `OmniTable` node that combines all of the node's instrumentation operations. Cursor objects contain a `filter` and an `output` clause; logically, a cursor inspects the execution instruction-by-instruction, producing the `output` whenever the `filter` is true. SteamDrill generates the filter clause of the cursor for each `OmniTable` in the stage by combining all selection (σ) and loop join instrumentation operators and generates the output clause using the output of the top-most projection (Π) instrumentation operator.

4.3.2 Materialization

Next, SteamDrill materializes `OmniTable` nodes by executing the cursor objects on top of a replay of the execution associated with the tables. It uses epoch parallelism [34, 35] to parallelize cursor evaluation. Epoch parallelism partitions a replay execution into time slices, called epochs. It assigns each epoch to a separate core in a compute cluster and uses checkpoints, generated during recording, so that each core executes each cursor over only its assigned epoch.

However, naive cursor evaluation (i.e., instrumenting every instruction) imposes a many orders of magnitude slowdown. So, SteamDrill analyzes the filter clause of each cursor to identify instructions at which the system can elide cursor evaluation to optimize performance. For example, SteamDrill identifies that the cursors in the second stage of Figure 8 only need to be evaluated at malloc and free and removes all other cursor evaluations. Our prototype identifies these optimizations by finding comparisons to the program counter.

Additionally, SteamDrill calculates operators in the stage that were not assigned as instrumentation operators for any `OmniTable` node (e.g., Merge Join_1 and Merge Join_2 in Figure 8). SteamDrill uses existing algorithms to calculate merge join and aggregation operators [1, 16]. Additionally, it executes the program associated with each `Generator` in the stage to calculate `Generator` operators.

5 Implementation

We implement our SteamDrill prototype on top of Spark [1] and Arnold [10]; below, we describe its key components.

Spark SQL. Our prototype introduces new relational operators and base tables for `OmniTables` and `Generators`. We added support for block-nested-loop joins, stored procedures, and polymorphic columns (§3) by serializing data to and from a JSON format. We added catalyst rules for our `OmniTable`-specific join order and algorithm preferences (§4.2.1). Each rule required 25 lines of code, so we expect that developers will be able to easily add custom rules as needed for their debugging workflows.

Instrumentation. Efficient cursor instrumentation plays a vital role in our prototype's performance. Debugging tools often use dynamic instrumentation frameworks (e.g., PIN [23]), which are a scalability bottleneck when SteamDrill parallelizes the replay execution across many cores [34]. Our prototype performs static binary instrumentation. It disassembles the application binaries and rewrites the basic blocks contained in the application to call cursors, as required for the breakpoints determined from each cursor. The system single-steps execution for cursors that do not produce breakpoints.

Time Column. The time column is a critical element of the `OmniTable` query model, but, deriving the column by counting all instructions or basic blocks would be too expensive. We observe that instructions progress from low to high, except in the case of a backwards control-flow (e.g., branch, call, or return instructions that jump to a program location at a lower address). Thus, our prototype uses the number of backwards control-flow operations as a the first element of the time column and breaks ties using the instruction pointer. Serendipitously, Intel provides deterministic performance counters for conditional branch and call instructions[2], which allow our prototype to compute the number of backwards control-flow operations by counting the number of unconditional backwards branches during execution and adding the value of these performance counters.

6 Evaluation

In this section, we evaluate the `OmniTable` query model and SteamDrill by answering the following questions: "Does the `OmniTable` query model improve upon existing debugging interfaces?", "Does SteamDrill accelerate debugging questions?", and "How do SteamDrill design decisions impact query performance?".

We perform 5 detailed case studies of how a developer could use an `OmniTable` and SteamDrill to solve real-world bugs from open-source servers (§6.1) from which we derive 14 debugging questions. We implement the debugging questions using `OmniTable` queries and `gdb`'s python bindings, which provide a python interface for traditional `gdb` features (e.g., breakpoints and backtraces). We compare the complexity of the 14 `OmniTable` queries and `gdb` scripts using metrics from the software engineering community (§6.2). We deploy SteamDrill on a CloudLab [37] cluster of 8 r320 machines (8-core Xeon E5-2450 2.1 GHz processor, 16 GB Ram, 10 Gbps NIC) to evaluate the performance for 3 representatives from the original 14 debugging questions (§6.3). We calculate the latency results below as the average over 10 trials and include 95% confidence intervals.

6.1 Case Studies

We performed 5 detailed case studies by identifying the debugging questions that a developer would ask when solving real-world bugs. We choose notoriously difficult bugs including livelock, intermittent performance problems, and atomicity violations (on average, the bugs in our study took 159 days from being opened to the commit that fixed the bug). We choose case studies from popular open-source applications: redis, Memcached, Apache, and Sqlite. The redis 4323 case study is described in §2; below, we describe case studies for debugging a livelock [28] and atomicity violation [27] in Memcached. We omit a description of a performance degradation in Apache [6] and a segmentation fault in Sqlite [41].

The case studies illustrate the benefits of the `OmniTable` query model along two key dimensions: first, the all-inclusive

[2] Note that most performance counters are not deterministic

```
1 Select f.Name, Count(*)
2 From Insts(ot) as i PrevJoin Funcs(ot) as f
3   On i.time, f.entryTime, i.thread=f.thread
4 Where f.exitTime=NULL
```

Listing 8: The First query for Memcached 271.

```
1 Select eip, True, False
2 From Vars(ot)
3 Where name="status"
4 Pivot Count() in (True, False)
```

Listing 9: The second query for Memcached 127.

state exposed by the table offers a powerful window into an execution's behavior. Second, SQL aggregations provide a powerful tool for summarizing and comparing program state. These features are particularly powerful when used in tandem. For example, in Memcached 271, the developer identifies the function that contains a livelock by counting the number of instructions executed by the functions left on the call stack at the end of the execution. This logic cannot be expressed in existing high-level debugging tools and is very complex when expressed using procedural tools such as `gdb`.

Memcached 271. In this case study, a developer observes livelock in the Memcached key-value store [28]. Livelock is notoriously difficult to diagnose since a developer needs to identify the cause of a missing property: forward progress [33].

In contrast, the `OmniTable` model allows the developer to succinctly track millions of execution events and use aggregations to identify anomalous execution state. Their first query, shown in Listing 8, identifies which function contains the livelock by counting the number of instructions executed during each function on each thread's call stack at program termination. The query matches each executed instruction with the most recent function called on the same thread to determine which function contained the instruction (Lines 2 and 3). It counts how many instructions were executed (Line 1) by each function that did not return (Line 4).

The output identifies a single function with a high number of branches. The function traverses a linked-list, which the developer suspects is corrupted. The developer's second query counts how many times each function that updates the linked list is called with every possible function argument value and shows a single anomalous call to free a linked-list item in which the item is still resident in the linked-list. Memcached reference counts linked-list items, so the developer's third and final query tracks all reference count updates and identifies an overflow that leads to the erroneous free of the item.

Memcached 127. This case study involves an atomicity violation in Memcached. An integer stored in the cache has the wrong value after all updates, which is challenging to debug since the developer does not know which program state to track or when to track it. Atomicity violation tools [32] use heuristics and may misidentify the root cause of the bug.

Query	Lines		Nodes		Halstead (s)	
	gdb	OT	gdb	OT	gdb	OT
Apache 60956 Q1	20	6	94	54	518	263
Apache 60956 Q2	30	9	113	122	989	1350
Memcached 127 Q1	7	4	48	39	147	82
Memcached 127 Q2	11	4	74	26	518	38
Memcached 271 Q1	35	3	149	26	1471	62
Memcached 271 Q2	12	4	69	23	397	34
memcached 271 Q3	10	3	45	26	140	39
redis 4323 Q1	17	3	74	23	529	45
redis 4323 Q2	7	3	24	31	35	65
redis 4323 Q3	22	3	113	83	930	757
redis 4323 Q4	33	5	147	112	1620	1033
redis 4323 Q5	7	3	19	19	17	19
sqlite 787fa71 Q1	22	10	110	77	911	520
sqlite 787fa71 Q2	41	8	151	96	1489	684
Average	20	5	88	54	694	357

Table 1: Lines, Nodes, and Halstead Complexity for debugging questions expressed using `OmniTable` queries (OT) and `gdb` python scripts (`gdb`).

The `OmniTable` model, particularly SQL aggregations, provide a powerful tool for comparing the state of their program at many points-in-time to identify anomalous program state. The developer first isolates the module that contains the error. In particular, they determine if the bug arises when initially parsing requests or when processing them by using a `count` aggregate to count the number of times the function at the boundary between parsing and processing is called with each possible set of arguments. The query shows that the problem arises when processing requests.

The processing code maintains a boolean variable, `valid`, that tracks the validity of a global pointer used by the code. The developer's second query, shown in Listing 9, identifies how often `valid` is set to `true` and `false` during each of the instruction within the processing logic. It uses a **`Pivot`** operator to produce a row for each instruction and show the number of times `valid` is set to `true` and `false` across all executions of the instruction. The second query identifies a few instructions at which status has an anomalous state. The anomalous instructions do not modify the status, so the developer concludes that another thread must modify the status and identifies a mistake in the processing logic's use of a mutex.

6.2 Complexity

We implemented the 14 debugging queries from our 5 case studies using `OmniTable` queries and implemented equivalent logic using `gdb` python scripts. Qualitatively, we observe that `OmniTable` queries are less complex due SQL aggregations, the all-inclusive nature of an `OmniTable`, and the structured approach provided by high-level views: `OmniTable` queries usually involve an aggregation after joining a few high level views, whereas imperative debugging scripts regularly use multi-dimensional data-structures to track state, nested control-flow to implement aggregations, and complex

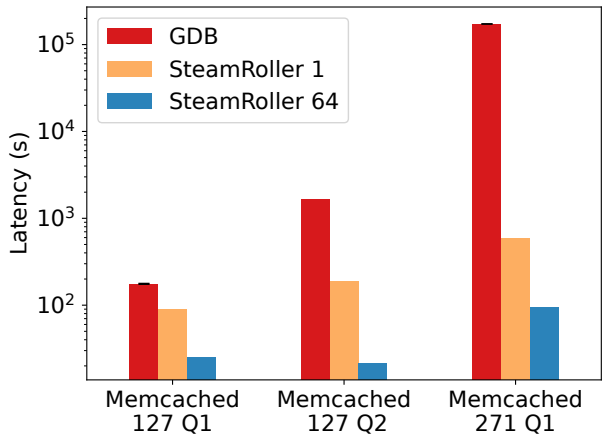


Figure 9: SteamDrill query latency on a single core and on 64 cores compared to `gdb` script latency (which is sequential). Y-axis is log-scale.

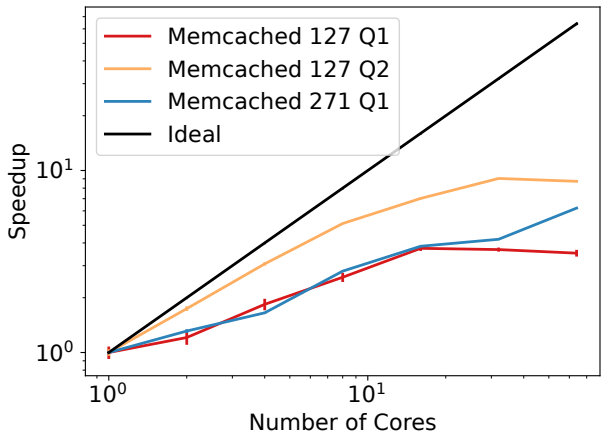


Figure 10: SteamDrill scalability. Shows number of cores on the x-axis vs. speedup on the y-axis; both axes are log-scale.

regular expressions to identify instrumentation points. We measure complexity of each `OmniTable` query and `gdb` script using three software engineering metrics: the number of lines of code, the number of terms in the abstract syntax tree (AST), and the Halstead complexity, which estimates the amount of time it would take to correctly produce the query or script using properties of the AST [18]. We included the definition of user-defined views (e.g., `DefinedMemory(ot)`) into the `OmniTable` queries that use them, so our results are an upper-bound on `OmniTable` query complexity.

Table 1 shows the results, indicating that `OmniTable` queries are less complex than `gdb` scripts. By geometric mean, `OmniTable` queries require 3.74 times fewer lines, 1.70 times fewer nodes, and 2.75 times less estimated time to develop than `gdb` scripts. There are only three queries that are more complex when expressed using the `OmniTable` model, the second and fifth redis 4323 queries, and the second Apache 60956 query. The two redis queries are small for both representations. The second Apache query suffers from the lack of kernel state in an `OmniTable`. The query identifies all blocking file descriptors, which requires substantial logic to track all function calls in the `OmniTable` model, but can be calculated in `gdb` using `fcntl`. Extending the `OmniTable` to include kernel state would reduce the complexity.

6.3 Query Latency

We evaluate the latency of `OmniTable` queries and `gdb` scripts for 3 representative queries from our case studies. We choose queries that use all of the high level views in our prototype (i.e., `Funcs(ot)`, `Vars(ot)`, and `Insts(ot)`) and offer a wide range of performance on current tools, from ~22 minutes to ~2 days. Figure 9 shows the latency of each debugging question evaluated using `gdb`, SteamDrill with a single core, and SteamDrill with 64 cores, with latency plotted on a log-scale. We executed Memcached 271 Q1 for 48 hours before killing the program and report its latency as 48 hours.

SteamDrill is significantly faster than `gdb`. SteamDrill query latency is between 2 and 290 times (with a geometric mean of 17) faster than `gdb` latency when using a single core, and between 6.9 and 1809 (with a geometric mean of 99) times faster than `gdb` latency when using 64 cores.

6.4 Optimizations

Next, we evaluate the impact of three optimizations on SteamDrill's latency: parallelization, multi-replay resolution, and performance-counters.

Scalability. We evaluate the query latency of SteamDrill queries when using 1–64 cores; Figure 10 shows the speedup on a log-log scale. SteamDrill queries are 10.5 times faster using 64 cores than when run sequentially. Importantly, whereas prior parallelization efforts require the developer to substantially redesign their debugging code [30, 34, 35, 38, 46], the parallelized and sequential `OmniTable` queries are identical. The current scalability bottlenecks are caused by high initialization and serialization cost in Spark and the high cost of compiling cursors.

Multi-Replay Resolution. We evaluate the impact of multi-replay query resolution on the Memcached 271 Q2 query. We calculate the query latency when using two rounds of replay (the approach chosen by the SteamDrill planner) and when using a single round of replay on 64 cores. SteamDrill is 3.6 times faster when using multi-replay resolution.

Performance Counters. We evaluate the impact of using performance counters to accelerate the calculation of the `time` column in an `OmniTable`. We executed the 3 queries with and

Tool	Model	Observations	Aggregates
Execution Mining [20]	Stream	All	No
Fay [12]	Stream	Partial	Partial
Pivot Tracing [24]	Relational	Log-Based	Partial
G2 [17]	Graph	Log-Based	Manual
PQL [25]	Stream	Partial	No
PTQL [14]	Relational	Partial	No
EndoScope [7]	Stream	Partial	Yes
EBBA [5]	Stream	Log-Based	No
TQuel [40]	Relational	Log-Based	Partial
`OmniTable`	Relational	Everything	Yes

Table 2: Feature comparison of high-level debugging tools .

without using performance counters (when disabled, SteamDrill instruments all `jump`, `call`, and `return` instructions) on 64 cores. The performance counter optimization accelerates query latency by a factor of 1.6.

7 Related Work

The `OmniTable` query model is the first debugging model that exposes all application state as a single entity and enables succinct observations via a high-level declarative language. Below we describe work related to high-level languages for debugging, using deterministic replay for debugging, and applying optimizations to accelerate debugging.

Existing systems support high-level debugging languages to reduce programming complexity; Table 2 illustrates the limitations of prior work compared to the `OmniTable` model. Execution Mining [20], PQL [25], EBBA [5] and EndoScope [7] expose a time-stream model of execution, which complicates debugging since it is difficult to summarize data over time (e.g., these tools cannot express the `Funcs(ot)` view since it contains execution data from multiple points-in-time). Other systems limit visibility of execution state: Fay [12], PQL [25], EBBA [5], EndoScope [7], and PTQL [14] expose partial program state consisting of only the function calls or global variables values in an execution. Pivot Tracing [24], G2 [17], EBBA [5], and TQuel [40] require manual instrumentation to enable observations, which essentially amounts to supporting queries over software logs. Finally, many tools provide no, or very few, aggregates [14, 20, 25]; G2 [17] supports aggregates but requires that they be expressed in terms of a graph processing language.

Many `OmniTable` queries compare correct execution behavior to incorrect execution behavior, similar to statistical debugging approaches [22]. There are two key differences (1) statistical bug isolation requires observing many correct and incorrect executions to come to a statistical verdict, whereas developers can often get a "sense" for correctness using an `OmniTable` query with fewer examples and (2) statistical debugging approaches hard code the values that they compare (e.g., function argument values), whereas developers can customize `OmniTable` queries to use program constructs best suited to their applications.

Many systems have noted that deterministic replay can be a great help when debugging software problems [8, 13, 19, 31, 42, 45]. Such systems enable a debugging program to explore an execution's time-sequence in reverse, but retain a procedural interface.

Recently, JetStream [34] and Sledgehammer [35] use deterministic replay as a vehicle for parallelizing debugging, which our prototype uses to accelerate `OmniTable` queries. However, these tools support procedural debugging models, similar to `gdb`, and consequently suffer from the programming complexity.

Existing tools do not decouple debugging logic's execution from the original execution to optimize query latency. PARTICLE [14], Fay [12], Pivot Tracing [24] and PMSS [21] reduce the debugging performance overhead using traditional SQL optimizations (e.g., predicate push-down). However, these tools add instrumentation to the program and re-execute it to recreate the bug, which tightly couples the execution of debugging and the original execution and increases performance overhead. Additionally, by inspecting new executions, these systems are cannot perform all SteamDrill performance optimizations, particularly multi-replay query resolution.

8 Conclusion

In this paper, we propose the `OmniTable` query model, a new debugging paradigm that reduces the programming complexity and performance overhead of debugging without restricting the execution state that a developer can observe. We show that the query model simplifies debugging questions compared to existing state-of-the-art tools by performing case studies of bugs reported in popular open-source software. Unfortunately, an `OmniTable`, the key abstraction in the model, cannot be stored or calculated due to its extreme size. So, our prototype, SteamDrill, implements *lazy materialization*: it delays an `OmniTable`'s calculation until a developer queries the table. It uses deterministic record and replay to store the execution associated with each `OmniTable` and then generates instrumentation and traces a new replay execution to resolve each developer query on-demand. The system uses declarative optimizations, debugging optimizations, and a novel multi-replay strategy to accelerate debugging queries by an order of magnitude compared to state-of-the-art tools.

9 Acknowledgements

We would like to thank our shepherd, Ding Yuan, and the anonymous reviewers for their insightful comments. The work was supported by the National Science Foundation under grant DGE-1256260.

References

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[3] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29thACM/IEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[4] Subarno Banerjee, David Devecsery, Peter M Chen, and Satish Narayanasamy. Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 490–504. IEEE, 2019.

[5] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995.

[6] Bug 60956. https://bz.apache.org/bugzilla/show_bug.cgi?id=60956.

[7] Alvin Cheung and Samuel Madden. Performance profiling with endoscope, an acquisitional software monitoring framework. *Proc. VLDB Endow.*, 1(1):42–53, aug 2008.

[8] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. Rept: Reverse debugging of failures in deployed software. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, OSDI'18, pages 17–32, 2018.

[9] David Devecsery, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 348–362, 2018.

[10] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, Broomfield, CO, October 2014.

[11] Marc Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, apr 1997.

[12] Ulfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 311–326, October 2011.

[13] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'07, pages 21–21, 2007.

[14] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 385–402, New York, NY, USA, 2005. ACM.

[15] Google sanitizers issues. https://github.com/google/sanitizers/issues?q=is%3Aissue+is%3Aopen+ASAN.

[16] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[17] Zhenyu Guo, Haoxiang Lin, Mao Yang, Dong Zhou, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011.

[18] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977.

[19] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 1–15, April 2005.

[20] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. Execution Mining. In *Proceedings of*

the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), March 2012.

[21] Yingsha Liao and Donald Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Transactions on Software Engineering*, 18(11):969–978, 1992.

[22] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, PLDI '05, page 15–26, New York, NY, USA, 2005. Association for Computing Machinery.

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[24] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015.

[25] Michael Martin, Benjamin Livshits, and Monica S Lam. Finding application errors and security flaws using pql: a program query language. *ACM SIGPLAN Notices*, 40(10):365–383, 2005.

[26] Steve McConnell. *Code complete*. Pearson Education, 2004.

[27] memcached - issue #127. https://code.google.com/archive/p/memcached/issues/127.

[28] Memcached gets a dead loop in func assoc_find. https://github.com/memcached/memcached/issues/271.

[29] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[30] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Seattle, WA, March 2008.

[31] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference*, Santa Clara, CA, July 2017.

[32] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2009.

[33] Rahul Patil and Boby George. Tools and techniques to identify concurrency issues. hhttps://docs.microsoft.com/en-us/archive/msdn-magazine/2008/june/tools-and-techniques-to-identify-concurrency-issue.

[34] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation*, Savannah, GA, November 2016.

[35] Andrew Quinn, Jason Flinn, and Michael Cafarella. Sledgehammer: Cluster-fueled debugging. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, pages 545–560, 2018.

[36] Redis 4.x lazyfree: memory leak may happen when free slowlog entry #4323. https://github.com/redis/redis/issues/4323.

[37] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[38] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2008.

[39] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, December 2009.

[40] Richard Snodgrass. Monitoring in a software development environment: A relational approach. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 124–131, New York, NY, USA, 1984. ACM.

[41] Assertion fault when multi-use subquery implemented by co-routine. https://www.sqlite.org/src/tktview/787fa71.

[42] Sudarshan Srinivasan, Christopher Andrews, Srikanth Kandula, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 29–44, Boston, MA, June 2004.

[43] Michael Stonebraker and Lawrence A Rowe. The design of postgres. *ACM Sigmod Record*, 15(2):340–355, 1986.

[44] Kde bugtracking system. https://bugs.kde.org/buglist.cgi?component=memcheck&product=valgrind&resolution=---.

[45] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, October 2011.

[46] Benjamin Wester, David Devescery, Peter M. Chen Jason Flinn, and Satish Narayanasamy. Parallelizing data race detection. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, TX, March 2013.

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, San Jose, CA, April 2012. USENIX Association.

XRP: In-Kernel Storage Functions with eBPF

Yuhong Zhong[1], Haoyu Li[1], Yu Jian Wu[1], Ioannis Zarkadas[1], Jeffrey Tao[1], Evan Mesterhazy[1], Michael Makris[1], Junfeng Yang[1], Amy Tai[2], Ryan Stutsman[3], and Asaf Cidon[1]

[1]Columbia University, [2]Google, [3]University of Utah

Abstract

With the emergence of microsecond-scale NVMe storage devices, the Linux kernel storage stack overhead has become significant, almost doubling access times. We present XRP, a framework that allows applications to execute user-defined storage functions, such as index lookups or aggregations, from an eBPF hook in the NVMe driver, safely bypassing most of the kernel's storage stack. To preserve file system semantics, XRP propagates a small amount of kernel state to its NVMe driver hook where the user-registered eBPF functions are called. We show how two key-value stores, BPF-KV, a simple B$^+$-tree key-value store, and WiredTiger, a popular log-structured merge tree storage engine, can leverage XRP to significantly improve throughput and latency.

1 Introduction

With the rise of new high performance memory technologies, such as 3D XPoint and low latency NAND, new NVMe storage devices can now achieve up to 7 GB/s bandwidth and latencies as low as 3 µs [11, 19, 24, 26]. At such high performance, the kernel storage stack becomes a major source of overhead impeding both application-observed latency and IOPS. For the latest 3D XPoint devices, the kernel's storage stack *doubles* the I/O latency, and it incurs an even greater overhead for throughput (§2.1). As storage devices become even faster, the kernel's relative overhead is poised to worsen.

Existing approaches to tackle this problem tend to be radical, requiring intrusive application-level changes or new hardware. Complete kernel bypass through libraries such as SPDK [82] allows applications to directly access underlying devices, but such libraries also force applications to implement their own file systems, to forgo isolation and safety, and to poll for I/O completion which wastes CPU cycles when I/O utilization is low. Others have shown that applications using SPDK suffer from high average and tail latencies and severely reduced throughput when the schedulable thread count exceeds the number of available cores [54]; we confirm this in §6, showing that in such cases applications indeed suffer a 3× throughput loss with SPDK.

In contrast to these approaches, we seek a readily-deployable mechanism that can provide fast access to emerging fast storage devices that requires no specialized hardware and no significant changes to the application while working with existing kernels and file systems. To this end, we rely on BPF (Berkeley Packet Filter [67, 68]) which lets applications offload simple functions to the Linux kernel [8]. Similar to kernel bypass, by embedding application-logic deep in the kernel stack, BPF can eliminate overheads associated with kernel-user crossings and the associated context switches. Unlike kernel bypass, BPF is an OS-supported mechanism that ensures isolation, does not lead to low utilization due to busy-waiting, and allows a large number of threads or processes to share the same core, leading to better overall utilization.

The support of BPF in the Linux kernel makes it an attractive interface for allowing applications to speed up storage I/O. However, using BPF to speed up storage introduces several unique challenges. Unlike existing packet filtering and tracing use cases, where each BPF function can operate in a self-contained manner on a particular packet or system trace — for example, network packet headers specify which flow they below to — a storage BPF function may need to synchronize with other concurrent application-level operations or require multiple function calls to traverse a large on-disk data structure, a workload pattern we call "resubmission" of I/Os (§2.3). Unfortunately the state required for resubmission such as access-control information or metadata on how individual storage blocks fit in the larger data structure they belong to is not available at lower layers.

To tackle these challenges, we design and implement XRP (eXpress Resubmission Path), a high-performance storage data path using Linux eBPF. XRP is inspired by XDP, the recent efficient Linux eBPF networking hook [28]. In order to maximize its performance benefit, XRP uses a hook in the NVMe driver's interrupt handler, thereby bypassing the kernel's block, file system and system call layers. This allows XRP to trigger BPF functions directly from the NVMe driver as each I/O completes, enabling quick resubmission of I/Os that traverse other blocks on the storage device.

The key challenge in XRP is that the low-level NVMe driver lacks the context that the higher levels provide. Those layers contain information such as who owns a block (file system layer), how to interpret the block's data, and how to traverse the on-disk data structure (application layer).

Our insight is that many storage-optimized data structures that power real-world databases [10, 12, 20, 27, 44, 66, 70, 80] – such as on-disk B-trees, log-structured merge trees, and log segments – are typically implemented on a small set of large files, and they are updated orders of magnitude less frequently than they are read; we validate this in §3. Hence, we exclusively focus XRP on operations contained within one file and on data structures that have a fixed layout on disk. Consequently, the NVMe driver only requires a minimal amount of the file system mapping state, which we term the *metadata digest*; this information is small enough that it can be passed from the file system to the NVMe driver so it can safely perform I/O resubmissions. This allows XRP to safely support some of the most popular on-disk data structures.

We present a design and implementation of XRP on Linux, with support for ext4, which can easily be extended to other file systems. XRP enables the NVMe interrupt handler to resubmit storage I/Os based on user-defined BPF functions.

We augment two key-value stores with XRP: BPF-KV, a B$^+$-tree based key-value store that is custom-designed for supporting BPF functions, and WiredTiger's log-structured merge tree, which is used as one of MongoDB's storage engines [27]. With random 512 B object reads on BPF-KV with multiple threads using a B$^+$-tree that has three index levels on disk, XRP has 47%–94% higher throughput and 20%–34% lower p99 latency than `read()`. XRP also enables more efficient sharing of cores among applications than kernel bypass: it is able to provide 56% better p99 latency than SPDK with two threads sharing the same core. In addition, XRP is able to consistently improve WiredTiger's performance by up to 24% under YCSB [41]. We open source XRP and our changes to BPF-KV and WiredTiger at `https://github.com/xrp-project/XRP`.

We make the following contributions.

1. **New Datapath**. XRP is the first datapath that enables the use of BPF to offload storage functions to the kernel.
2. **Performance**. XRP improves the throughput of a B-tree lookup by up to 2.5× compared to normal system calls.
3. **Utilization**. XRP provides latencies that approach kernel bypass, but unlike kernel bypass, it allows cores to be efficiently shared by the same threads and processes.
4. **Extensibility**. XRP supports different storage use cases, including different data structures and storage operations (e.g., index traversals, range queries, aggregations).

2 Background and Motivation

In this section we show why the Linux kernel is becoming a primary bottleneck with fast NVMe devices and provide a primer on BPF.

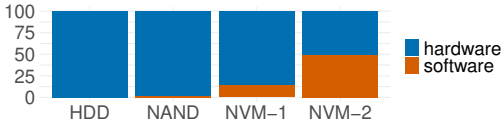


Figure 1: Kernel's latency overhead with 512 B random reads. HDD is Seagate Exos X16, NAND is Intel Optane 750 TLC NAND, NVM-1 is first generation Intel Optane SSD (900P), and NVM-2 is second generation Intel Optane SSD (P5800X).

kernel crossing	351 ns	5.6%
read syscall	199 ns	3.2%
ext4	2006 ns	32.0%
bio	379 ns	6.0%
NVMe driver	113 ns	1.8%
storage device	3224 ns	51.4%
total	6.27 µs	100.0%

Table 1: Average latency breakdown of a 512 B random `read()` syscall using Intel Optane P5800X.

2.1 Software is Now the Storage Bottleneck

New media like 3D Xpoint [1] and low-latency NAND [26], have led to new NVMe storage devices that exhibit single-digit µs latencies and millions of IOPS [11, 19, 24, 26]. The kernel storage stack is becoming a major performance bottleneck when accessing these devices. Figure 1 shows the percentage of time spent in the Linux stack when issuing a 512 B random read I/O on different storage devices. While the software overhead for the first generation of fast NVMe devices (first generation Intel Optane or Z-NAND) was non-negligible (~15%), with the latest generation of devices (Intel Optane SSD P5800X) the software overhead accounts for about half of the latency of each read request. The kernel's relative overhead will only get worse as storage devices become even faster.

Where is the time going? Table 1 shows the time spent in the different storage layers when issuing a random 512 B read with O_DIRECT on Optane P5800X. The experimental setup, which is used throughout this section, is a server with 6-core i5-8500 3 GHz with 16 GB of memory, using Ubuntu 20.04, and Linux 5.8.0. We also disable processor C-states and turbo boost, use the maximum performance governor, and disable KPTI [30]. The experiment shows that the most expensive layer is the file system (ext4), followed by the block layer (bio) and the kernel crossing, and that the total software overhead accounts for 48.6% of the average latency.

Why not just bypass the kernel? One approach to eliminate kernel overhead is to bypass it altogether [7, 65, 82, 83], leaving just the cost to post a request to the NVMe driver and the device's latency. However, kernel bypass is no panacea: each user is entrusted with full access to the device; they must also construct their own user space file systems [73, 74]. This means that there is no mechanism to enforce fine-grained

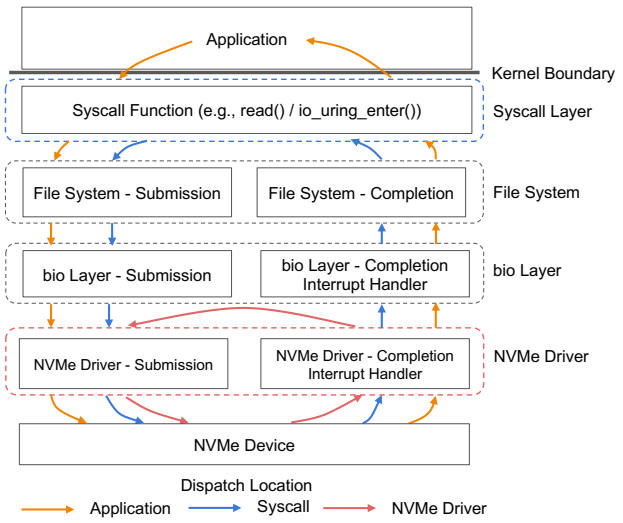


Figure 2: Dispatch paths for the application and two kernel hooks.

isolation or to share capacity among different applications accessing the same device. In addition, there is no efficient way for user space applications to receive interrupts on I/O completions, so applications must directly poll on device completion queues to obtain high performance. Consequently, when I/O is not the bottleneck, cores cannot be shared among processes, which results in significant under-utilization and wasted CPU. Furthermore, when more than one polling thread shares the same processor, the CPU contention between them coupled with the lack of synchronization lead all polling threads to experience degraded tail latency and significantly lower overall throughput. Recent work has highlighted this issue [54] and we reproduce it in §6.2.

2.2 BPF Primer

BPF (Berkeley Packet Filter) is an interface that allows users to offload a simple function to be executed by the kernel. Linux's framework for BPF is called eBPF (extended BPF) [23]. Linux eBPF is commonly used for filtering packets (e.g., TCPdump) [5, 6, 28, 52], load balancing and packet forwarding [5, 18, 25, 60], tracing [2, 4, 50], packet steering [46], network scheduling [53, 58] and network security checks [15]. Functions are verified by the kernel at install-time to ensure they are safe; for example, they are checked to make sure they do not contain too many instructions, unbounded loops, or accesses to out-of-bounds memory addresses [29]. After verification, which typically takes a few seconds or less, the eBPF functions can be called normally.

2.3 The Potential Benefit of BPF

BPF can be a mechanism for avoiding data movement between the kernel and user space in cases when a logical lookup requires a sequence of "auxiliary" I/O requests that generate intermediate data not needed directly by the application, such as in pointer-chasing workloads. For example, to traverse a B-tree index, a lookup at each level traverses the kernel's entire storage stack only to be thrown away by the application once it obtains the pointer to the next child node in the tree. Instead of a sequence of system calls from user space, each of the intermediate pointer lookups could be executed by a BPF function, which would parse the B-tree node to find the pointer to the relevant child node. The kernel would then submit the I/O to fetch the next node. Chaining a sequence of such BPF functions could avoid the cost of traversing kernel layers and moving data to user space.

	Latency	Speedup	Throughput	Speedup
User Space	78 µs	1×	109K IOPS	1×
Syscall Layer	68 µs	1.15×	130K IOPS	1.2×
NVMe Driver	40 µs	1.95×	276K IOPS	2.5×

Table 2: Average latency and throughput improvement with respect to user space when resubmitting I/O from the given layer; for kernel layers, resubmission is executed with a BPF function. Results shown for lookups on an on-disk B-tree of depth 10 [85].

Other popular on-disk data structures, such as log-structured merge trees (LSM trees) [70], also have such auxiliary pointer lookups which can be accelerated using BPF functions. Other types of operations that would benefit from such an approach include range queries, iterators, and other types of aggregations (e.g., obtain the maximum or average value in a range of key-value pairs). In all of these cases, only a single result or a small subset of the objects that might be accessed by the storage system ultimately need to be returned to the application.

The BPF function that resubmits (dispatches) I/O in auxiliary I/O workloads could be placed at any layer of the kernel. Figure 2 shows the I/O paths for both normal user space dispatch and for two possible locations of BPF resubmission hooks: in the syscall layer and in the NVMe driver. Zhong et al. [85] compared the performance improvement from a resubmission hook in both locations on workloads with auxiliary I/O by measuring the speedup of lookup queries on an on-disk B-tree of depth 10. The baseline for comparison is reading I/O through the `read` system call. Table 2 summarizes the results.

Best Case Acceleration. Dispatching the I/O requests from the NVMe driver provides a significant latency reduction (up to 49%) and corresponding speedup (up to 2.5×), since it bypasses almost the entire kernel software stack. On the other hand, as expected, issuing the BPF functions from the syscall dispatch layer only provides a maximum speedup of 1.25×, since the requests only benefit from eliminating kernel boundary crossings, which only account for 5-6% of the kernel overhead (Table 1). After reaching CPU saturation, the computation savings of reissuing the submissions from the NVMe driver translate into throughput improvements of 1.8-2.5×, depending on the number of threads in the workload [85].

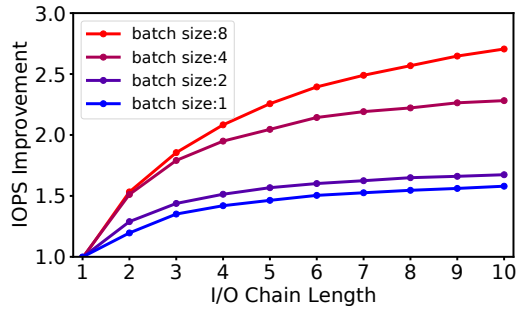


Figure 3: Single-threaded lookups with io_uring syscall, using NVMe driver hook.

Placing an eBPF hook *anywhere* in the kernel may improve throughput between 1.2–2.5×. However, pushing the I/O dispatching as close as possible to the storage device dramatically improves the performance of a traversal. Hence *to obtain the highest possible speedup, XRP's resubmission hook should reside in the NVMe driver*.

What about io_uring? io_uring is a new Linux system call framework [9] that allows processes to submit batches of asynchronous I/O requests, which amortizes user-kernel crossings. However, each I/O submitted with io_uring still passes through all the layers shown in Table 1, so each individual I/O still incurs the full storage stack overhead. In fact, BPF I/O resubmissions are largely complementary to io_uring: io_uring can efficiently submit batches of I/Os that trigger different I/O chains managed by BPF in the kernel.

Figure 3 shows throughput improvements when using io_uring with a BPF hook in the NVMe driver. I/O Chain Length denotes the total number of I/Os, including the initial I/O and the resubmitted I/Os. Figure 3 shows that BPF can increase throughput with respect to io_uring by up to 1.5× for small batch sizes and up to 2.5× as batch sizes increase.

In summary, BPF can benefit both legacy read and io_uring system calls. By placing the hook in the kernel NVMe driver, BPF may increase throughput of both legacy I/O and single-threaded io_uring by up to 2.5×.

3 Design Challenges and Principles

As shown in the previous section, I/O resubmission must occur as close to the device as possible in order to reap the greatest benefits. In the NVMe software stack, this is the NVMe interrupt handler. However, executing the resubmissions from within the NVMe interrupt handler, which lacks the context of the file system layer, introduces two major challenges.

Challenge 1: address translation and security. The NVMe driver has no access to file system metadata. In the example of an index traversal, XRP issues a read I/O to a particular block and executes a BPF function that would extract the offset of the next block it would like to query. However, this offset is meaningless to the NVMe layer, since it cannot tell which physical block the offset corresponds to without having access to the file's metadata and extents. Even if the application developer made the effort to embed physical block addresses to avoid the translation of the file system offset, which would be burdensome, the BPF function could access *any* block on the device, including blocks that belong to a file that the user does not have permissions to access.

Challenge 2: concurrency and caching. It is challenging to enable concurrent reads and writes issued from the file system with XRP. A write issued from the file system will only be reflected in the page cache, which is not visible to XRP. In addition, any writes that modify the layout of the data structure (e.g., modify the pointers to the next block) that are issued concurrently to read requests could lead XRP to accidentally fetch the wrong data. Both of these could be addressed by locking, but accessing locks from within the NVMe interrupt handler may be expensive.

Observation: most on-disk data structures are stable. Both of these challenges would make it difficult to implement arbitrary concurrent BPF storage functions. However, we make the observation that the files of many storage engines (e.g., LSM trees and B-trees) remain relatively stable. Some data structures simply do not modify on-disk storage structures in-place. For example, once an LSM tree writes its index files (called SSTables) to disk, they are immutable until they are deleted [12, 27, 44]. Accessing these immutable on-disk storage structures requires less synchronization effort. Similarly, even though some on-disk B-tree index implementations support in-place updates, their file extents remain stable for long periods of time. We verify this in a 24-hour YCSB [41] (40% reads, 40% updates, 20% inserts, Zipfian 0.7) experiment on MariaDB running TokuDB [20], which uses a fractal tree (an on-disk B-tree variant) as its lookup index. We found the index file's extents only changed every 159 seconds on average, with only 5 extent changes in 24 hours unmapping any blocks, making it possible to cache file system metadata in the NVMe driver without the overhead of frequent updates. We also make the observation that in all of these storage engines, the indices are stored on a small number of large files, and each index does not span multiple files.

Design principles. These observations and experiments inform the following design principles.

- **One file at a time.** We initially restrict XRP to only issue chained resubmissions on a single file. This greatly simplifies address translation and access control, and it minimizes the metadata that we need to push down to the NVMe driver (the *metadata digest*, §4.1.3).
- **Stable data structures.** XRP targets data structures, whose layout (i.e. pointers) remain immutable for a long period of time (i.e. seconds or more). Such data structures include the indices used in many popular commercial storage engines, such as RocksDB [44], LevelDB [12], TokuDB [12] and WiredTiger [27]. Since the cost of im-

plementing locks in the NVMe layer is high, we also initially do not plan to support operations that require locks during the traversal or iteration of data structures.

- **User-managed caches.** XRP does not interface with the page cache, so XRP functions cannot safely be run concurrently if blocks are buffered in the kernel page cache. This constraint is acceptable since popular storage engines often implement their own user space caches [20,27,39,44]; Commonly they do this to fine-tune their caching and prefetching policies and to cache data in an application-meaningful way (e.g., cache key-value pairs or database rows instead of physical blocks).
- **Slow path fallback.** XRP is best-effort; if a traversal fails for some reason (e.g., the extent mappings become stale), the application must retry or fall back to dispatching the I/O requests using user space system calls.

4 XRP Design and Implementation

This section presents XRP's design and implementation with Linux eBPF and ext4. We describe the kernel modifications that enable XRP's resubmission logic in the interrupt handler, and how applications are modified to use XRP. We also discuss XRP's synchronization and scheduling limitations.

4.1 Resubmission Logic

The core of XRP augments the NVMe interrupt handler with resubmission logic that consists of a BPF hook, a file system translation step, and the construction and resubmission of the next NVMe request at the new physical offsets (Figure 4). Our modifications to the Linux kernel consist of ~900 lines of code: ~500 lines for the BPF hook and the changes to the NVMe driver, ~400 lines for the file system translation step.

When an NVMe request completes, the device generates an interrupt that causes the kernel to context switch into the interrupt handler. For each NVMe request that is completed in the interrupt context, XRP calls its associated BPF function (`bpf_func_0` in Figure 4), the pointer of which is stored in a field in the kernel I/O request struct (i.e. `struct bio`). After calling the BPF function, XRP invokes the metadata digest, which is usually a digest of file system state that enables XRP to translate the logical address of the next resubmission. Finally, XRP prepares the next NVMe command resubmission by setting the corresponding fields in the NVMe request, and it appends the request to the NVMe submission queue (SQ) for that core.

For a particular NVMe request, the resubmission logic is called as many times as necessary for subsequent completions as determined by the specific BPF function registered with the NVMe request. For example, for traversing a tree-like data structure, the BPF function would resubmit I/O requests for branch nodes and end resubmission whenever a leaf node is found. In our current prototype there is no hard limit on the number of resubmissions before the completion returns control to the application; such a limit would be necessary to prevent unbounded execution. A hard limit can be enforced by maintaining a resubmission counter in each I/O request descriptor. Since I/O request descriptors cannot be accessed from user space or from XRP's BPF programs, their hard resubmission limits cannot be overridden by users even if XRP has multiple BPF functions that execute request resubmissions. BPF function contexts are per-request, while the metadata digest is shared across *all* invocations of the interrupt handler across all cores. Safe concurrent access to the metadata digest relies on read-copy-update (RCU) (§4.1.3).

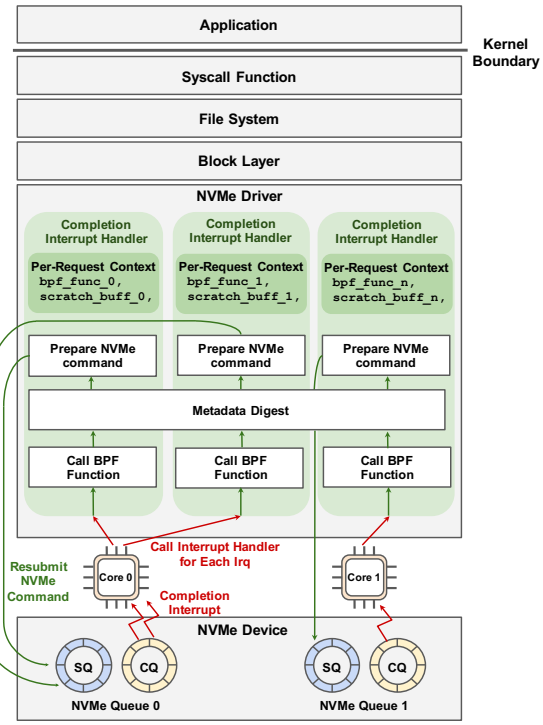


Figure 4: XRP architecture.

```
struct bpf_xrp {
  // Fields inspected outside BPF
  char *data;
  int done;
  uint64_t next_addr[16];
  uint64_t size[16];
  // Field for BPF function use only
  char *scratch;
};
uint32_t BPF_PROG_TYPE_XRP(struct bpf_xrp *ctxt);
```

Listing 1: Signature of BPF programs that can be loaded by XRP.

4.1.1 BPF Hook

XRP introduces a new BPF type (`BPF_PROG_TYPE_XRP`) with the signature shown in Listing 1 – any BPF function that matches the signature can be called from the hook. §5 presents one concrete BPF function matching this signature that is used in our application. For example, for on-disk data structure

traversal, the BPF function typically contains logic to extract the next offset to fetch from the block.

`BPF_PROG_TYPE_XRP` programs require a context with five fields, categorized into fields that are inspected or modified by the BPF caller (resubmission logic in the interrupt handler), and fields that should be private to the BPF function. Fields that are accessed externally include `data`, which buffers data read from the disk (e.g., a B-tree page waiting to be parsed by the BPF function). `done` is a boolean that notifies the resubmission logic whether to return to the user or continue resubmitting I/O requests. `next_addr` and `size` are arrays of logical addresses and their corresponding sizes that indicate the next logical addresses for resubmission.

In order to support data structures with fanout, multiple `next_addr` values can be supplied. By default we limit fanout to 16; on-disk data structures align their components to small multiples of device pages, so we have not encountered a need for higher fanout per completion. For example, chained hash table buckets are likely implemented as a chain of individual physical pages and the elements of an on-disk linked list are likely implemented at the granularity of physical pages. Setting a corresponding size field to zero issues no I/O.

`scratch` is a scratch space that is private to the user and the BPF function. It can be used to pass the parameters from the user to the BPF function. Also, the BPF function can use it to store intermediate data in between I/O resubmissions and to return data to the user. For example, in the first BPF invocation, the application can store a search key in the scratch buffer so that the BPF function can compare it with the keys in the disk block in order to find the next offset. When the I/O chain reaches the leaf node of the B-tree, the BPF function then places the key-value pair in the scratch buffer to return it back to the application. For simplicity, we assume that the size of the scratch buffer is always 4 KB. We find that a 4 KB scratch buffer is sufficient to support a BPF function for a production key-value store (§5). BPF functions can also use BPF maps to store more data if their intermediate data cannot fit into the scratch buffer. Each BPF context is private to one NVMe request, so no locking is needed when working with BPF context state. Letting the user supply a scratch buffer (instead of using BPF map) avoids the overhead of processes and functions having to call `bpf_map_lookup_elem` to access the scratch buffer.

4.1.2 BPF Verifier

The BPF verifier ensures memory safety by tracking the semantics of the value stored in each register [14]. A valid value can either be a scalar or a pointer. `SCALAR_TYPE` represents a value that cannot be dereferenced. The verifier defines various pointer types; most of them include extra constraints beyond the no out-of-bound access requirement. For example, `PTR_TO_CTX` is the type for the pointer to a BPF context. It can only be dereferenced using a constant offset so the verifier can identify which context field a memory operation

```
void update_mapping(struct inode *inode);
void lookup_mapping(struct inode *inode,
                    off_t offset, size_t len,
                    struct mapping *result);
```

Listing 2: Metadata digest: XRP exposes an interface to share logical-to-physical-block mappings between the file system and the IRQ handler.

accesses. Each BPF function type also defines a callback function `is_valid_access()` to perform additional checks on context accesses and to return the value type of the context field. `PTR_TO_MEM` describes a pointer referring to a fixed-size memory region. It supports dereferencing using a variable offset as long as the access is always within bounds. The `data` and `scratch` fields of the `BPF_PROG_TYPE_XRP` context are `PTR_TO_MEM` and the rest are `SCALAR_TYPE`. We augment the verifier to allow the `BPF_PROG_TYPE_XRP`'s `is_valid_access()` callback to pass the size of the data buffer or scratch buffer to the verifier so that it can perform the boundary check. We discussed our proposed modification to the verifier with the Linux eBPF maintainers, and they think it is sensible.

4.1.3 The Metadata Digest

In the conventional storage stack, the logical block offsets in on-disk data structures are translated by the file system in order to identify the next physical block to read. This translation step also enforces access control and security, preventing reading in regions that are not mapped to the open file. In XRP, the next logical address for a lookup is given by the `next_addr` field after the BPF call. However, translating this logical address to a physical address is challenging since the interrupt handler has no notion of a file and does not perform physical address translation.

To solve this, we implement the metadata digest, a thin interface between the file system and the interrupt handler that lets the file system share its logical-to-physical-block mappings with the interrupt handler, enabling safe eBPF-based on-disk resubmissions. The metadata digest consists of two functions (Listing 2). The update function is called within the file system when the logical-to-physical mapping is updated. The lookup function is called within the interrupt handler; it returns the mapping for a given offset and length. The lookup function also enforces access control by preventing BPF functions from requesting resubmissions for blocks outside of the open file. The inode address of the open file is passed to the interrupt handler in order to query the metadata digest. If an invalid logical address is detected, XRP returns to user space immediately with an error code. The application can then fall back to normal system calls to attempt its request again.

These two functions are specific to each file system, and even for a particular file system, there may be multiple ways to implement the metadata digest, presenting a tradeoff between ease of implementation and performance. For example, in

our implementation for ext4, the metadata digest consists of a cached version of the extent status tree, which stores the physical-to-logical block mappings. This cached tree is accessed by the update and lookup function of the interface, and it uses read-copy-update (RCU) for concurrency control. RCU enables the lookup function to be lockless and fast (96 ns on average).

To keep the cached tree up-to-date with the extents in ext4, the update function is called in two places in ext4: whenever extents are inserted or removed from the main extent tree. To prevent a race condition where an extent is modified while there is an inflight read on it, we maintain a version number for each extent to track its changes. After data is read, but before it is passed to the BPF function, a second metadata digest lookup is performed. If the corresponding extent no longer exists or its version number has changed, XRP will abort the operation. Since application-level synchronization usually prevents concurrent modifications and lookups on the same region of a file at the same time, version mismatches should only occur if the application is buggy or malicious.

An alternative, simpler implementation of the metadata digest for ext4 could simply pass through to existing update and access functions of the extent tree in ext4. In this case, the update function would be a no-op, because ext4 already keeps its extent tree up-to-date. However, such an implementation would be much slower on the lookup path, because the extent lookup function in ext4 acquires a spinlock, which would be prohibitively expensive in the interrupt handler.

For now, XRP only supports the ext4 file system, but the metadata digest can be easily implemented for other file systems. For example, in f2fs [64], logical-to-physical-block mappings are stored in the node address table (NAT). Similar to the ext4 implementation, an implementation of its metadata digest could cache a local copy of the NAT, which would be consulted in `lookup_mapping`. Then `update_mapping` would need to be called anywhere in f2fs where the NAT is updated.

4.1.4 Resubmitting NVMe Requests

After looking up the physical block offsets, XRP prepares the next NVMe request. Because this logic occurs in the interrupt handler, to avoid the (slow) `kmalloc` calls needed to prepare NVMe requests, XRP reuses the existing NVMe request struct (i.e. `struct nvme_command`) of the just-completed request. XRP simply updates the physical sector and block addresses of the existing NVMe request to the new offsets derived from the mapping lookup. Reusing NVMe request structs for immediate resubmission is safe because neither user space nor XRP BPF programs can access the raw NVMe request structs.

While `struct bpf_xrp` supports a maximum fanout of 16, in the current implementation a resubmitted I/O request can only fetch as many physical segments as the initial NVMe request. For example, if an initial NVMe request only fetches a single block, then all subsequent resubmissions for that request can only fetch a single physical segment. During a resubmission chain, if the BPF call returns multiple valid addresses in `next_addr`, XRP will abort the request. This limitation can be worked around by allocating and setting up 16 dummy NVMe commands in the first I/O request so that subsequent resubmissions can express fanout if necessary.

4.2 Synchronization Limitations

BPF currently only supports a limited spinlock for synchronization. The verifier only allows BPF programs to acquire one lock at a time, and they must release the lock before returning. Also, user space applications do not have direct access to these BPF spinlocks. Instead, they must invoke the `bpf()` syscall; the syscall can read or write the lock-protected structure while holding the lock for the duration of that operation. Hence, complex modifications that require synchronizing across multiple reads and writes cannot be accomplished in user space.

Users can implement custom spinlocks using BPF atomic operations. This allows both BPF functions and user space programs to acquire any spinlock directly. However, the termination constraint prohibits BPF functions from spinning to wait for a spinlock infinitely. Another option for synchronization is RCU. Since XRP BPF programs are run in the NVMe interrupt handler, which cannot be preempted, de-facto they are already in an RCU read-side critical section.

4.3 Interaction with Linux Schedulers

Process scheduler. Interestingly, we observed that a microsecond-scale storage device like Optane SSD interferes with Linux's CFS when multiple processes share the same core, *even when all I/O is issued from user space*. For example, in the case where an I/O-heavy and compute-heavy process share the same core, the I/O interrupts generated by the I/O-heavy process will be handled in the timeslice of the compute-heavy process. This may cause the compute-heavy process to be starved of CPU; in the worst case in our experiments, the compute-heavy process only received about 34% of what would be a "fair" allocation of CPU time. We experimentally verified this does not occur when using a slower storage device, which generates interrupts much less frequently. While XRP exarcerbates this problem by generating chains of interrupts, this issue is not specific to eBPF, and can also be caused by network-driven interrupts [59]. We leave this problem for future work.

I/O scheduler. XRP bypasses Linux's I/O scheduler, which sits at the block layer. However, the noop scheduler is already the default I/O scheduler for NVMe devices, and the NVMe standard supports arbitration at hardware queues if fairness is a requirement [17].

5 Case Studies

To use XRP, applications use the interface shown in Listing 3. Applications call libbpf [13] function `bpf_prog_load` to load a BPF function of type `BPF_PROG_TYPE_XRP` to be offloaded

```
int bpf_prog_load(const char *file,
                  enum bpf_prog_type type,
                  struct bpf_object **pobj,
                  int *prog_fd);
int read_xrp(int fd, void *buf, size_t count,
             off_t offset, int bpf_fd,
             void *scratch);
```

Listing 3: The XRP application interface consists of a libbpf function to load BPF functions into the kernel and a read syscall that requests that a BPF function be used. `bpf_prog_load` is an existing function in libbpf. `bpf_prog_load` returns a file descriptor for the loaded function, which must be passed to `read_xrp`. `read_xrp` adds two arguments to the standard `pread` [21] syscall: this file descriptor and a pointer to a 4 KB scratch space that is passed to the BPF context.

in the driver and call `read_xrp` to apply a specific BPF function to the read request. Applications can load multiple BPF functions with XRP. For example, a database can load a function for filtering and calculating aggregations from values on-disk and a function for `GET` point lookups. XRP allows the application to load multiple BPF functions into the kernel and to specify the BPF function to use in each `read_xrp` syscall. We present two case studies on how applications should be modified to use XRP.

5.1 BPF-KV

We built a simple key-value store, called BPF-KV, with which we can evaluate XRP against other baselines: Linux's synchronous and asynchronous system calls and kernel bypass (SPDK [82]). BPF-KV is designed to store a large number of small objects and to provide good read performance even under uniform access patterns. BPF-KV uses a B^+-tree index to find the location of objects, and the objects themselves are stored in an unsorted log. For simplicity, BPF-KV uses fixed-sized keys (8 B) and values (64 B). The index and the log are both stored in one large file. The index nodes use a simple page format with a header followed by keys followed by values. Leaf nodes contain a file offset pointing to the next leaf node, enabling efficient index traversal for range queries and aggregation. Object sizes are fixed, so updates occur in-place in the unsorted log. Newly inserted items are appended to the log; their index is initially stored in an in-memory hash table. Once the hash table fills, BPF-KV merges it with the on-disk B^+-tree file.

Caching. BPF-KV implements a user space DRAM cache for index blocks and objects. To reduce the number of I/Os it needs to issue for lookups, BPF-KV caches the top k levels of the B^+-tree index. With a sufficiently large number of objects, it is not possible to fit the entire index in the cache. Consider the case where BPF-KV is used to store 10 billion 64 B objects. In BPF-KV's index, each node is 512 B (matching the access granularity of the Optane SSD); hence, the tree has a fanout of 31 (i.e. each internal node can store pointers to 31 children). Therefore, 10 billion objects would require an index with 8 levels. Fitting 6 index levels in DRAM is expensive and would require 14 GB, while fitting 7 levels or more becomes prohibitively expensive (437 GB of DRAM or more). So, to support a large number of keys, BPF-KV would require at the minimum 3-4 I/Os from storage for each lookup, including a final I/O to fetch the actual key-value pair from disk. Also note that having a hard memory budget for caching the index is common in many real-world key-value stores (e.g., RocksDB [45], DocumentDB [78], SplinterDB [40], TokuDB [20]), since the index cache often competes with other parts of the system that need memory, such as filters and the object cache.

BPF-KV also maintains a least recently used (LRU) object cache of the most popular key-value pairs. Before looking up an object on disk, BPF-KV first checks whether it is stored in the object cache. If not, it checks whether it is indexed in the in-memory hash table. If the item is not found in the in-memory hash table, it looks up the object by accessing the first k cached levels of the index. Once it encounters an index node that is not cached, it completes the index and the final lookup on disk.

To find an object without XRP, BPF-KV traverses the B-tree until the desired value is found using an I/O request per level. For example, if the index contains 7 levels and the first 3 are cached and read from DRAM, then the traversal will issue 4 I/Os to navigate the rest of the tree, followed by a final I/O to fetch the object from the log.

BPF function. Listing 4 shows the BPF function used in BPF-KV to lookup a key-value pair. We omit the code to handle the final lookup in the log for simplicity. `struct node` defines the layout of B^+-tree index nodes whose size is 512 B. The BPF function `bpfkv_bpf` first extracts the target key stored in the scratch buffer, and then it linearly searches the slots in the current node to find the next node to read.

Interface modifications. We replace `read` calls with `read_xrp`. Before calling into `read_xrp`, BPF-KV first allocates a buffer for the scratch space and calculates the offset at which to start the lookup.

Range queries. BPF-KV supports range queries returning a variable number of objects. We implement a BPF function that runs as a state machine, allowing the operation to be suspended and resumed when objects are returned to the application for processing. The BPF function state, including the beginning and end of the range, and the retrieved objects, are stored in the scratch space (up to 32 72-byte key-value pairs). On the initial invocation, the function traverses to the leaf node that contains the starting key. Once the first key in the range is found, the function stores the leaf node in the scratch space and requests the block containing the corresponding value. On the next BPF invocation, the function stores the value in the scratch space and it continues the index scan

```
struct node {
  uint64_t num; uint64_t type;
  uint64_t key[31]; uint64_t ptr[31];
};
uint32_t bpfkv_bpf(struct bpf_xrp *ctxt) {
  uint64_t key = *((uint64_t*)ctxt->scratch);
  struct node *n = (struct node *)ctxt->data;
  if (n->type == LEAF_NODE) {
    ctxt->done = true;
    return 0;
  }
  int i;
  for (i = 1; i < n->num; i++)
    if (key < n->key[i]) break;
  ctxt->done = false;
  ctxt->next_addr[0] = n->ptr[i - 1];
  ctxt->size[0] = 512;
  return 0;
}
```

Listing 4: BPF function for BPF-KV.

on the cached leaf node. When the leaf node has been read completely, the function submits a request for the next leaf node using the node's next-leaf file offset. The function returns to the application in three cases: 1) the function reaches a key past the end of the range; 2) the function reaches the end of the index; 3) the function fills the scratch space with values read from the log. In the last case, the application can process the values and re-invoke the BPF function with the range query state, allowing the range query to resume from where it left off.

Aggregations. BPF-KV also supports aggregation operations, such as `SUM`, `MAX` and `MIN`. We implement these operations on top of the BPF range query function by setting a bit that causes the function to perform the corresponding aggregation instead of returning the individual values. Since aggregation queries return a single answer, storing values in the scratch space does not limit the number of I/O resubmits the BPF function can request.

5.2 WiredTiger

WiredTiger is a popular key-value store that is the default backend for MongoDB [27]. We use it as a case study since it is a relatively simple and open key-value store that is used in production. WiredTiger provides an option to use an LSM tree where data is split into different levels; each level contains a single file. Each file uses a B-tree index with the key-value pairs embedded in the tree's leaf nodes. The files are read-only; updates and inserts are written into a buffer in memory. When the buffer is full, the data is written out in a new file. We configure the B-tree page size to be the same as our Optane SSD's block size (512 B). Our modification to WiredTiger is around 500 lines of code, which mainly consist of buffer allocation, extending function signatures and wrapping the XRP syscall. XRP helps accelerate reads that are serviced from disk, and it does not affect updates or inserts, which are always absorbed by WiredTiger's in-memory buffer.

BPF function. To use XRP, WiredTiger installs a BPF function similar to the one shown in Listing 4. The difference is in order to find the next lookup address from the current page, the BPF function contains a port of WiredTiger's B-tree page parsing code. This parsing logic replaces the `for` loop in Listing 4.

The WiredTiger BPF function also makes several modifications to make the BPF program compile correctly and pass the BPF verifier. The modifications mainly consist of adding bounds on loops to avoid infinite loops, masking pointers to eliminate out-of-bound access, and initializing local variables to prevent access to uninitialized registers. We also use the BPF function-by-function verification feature [3] to break a complex function into several simple sub-functions. This allows BPF functions to be verified independently, so the functions that have been verified do not need another round of verification when being called by other functions. The function-by-function verification feature also supports more complex BPF programs without exceeding the verifier's restrictions on function length.

Caching. WiredTiger maintains a least recently used (LRU) cache for its B-tree internal pages and leaf pages. When looking up a new key-value pair, WiredTiger caches the entire lookup path including the leaf page in the cache. In order to comply with WiredTiger caching semantics, the BPF function described in the previous section also returns all traversed pages so that WiredTiger can cache them. The BPF function stores traversed pages in the scratch buffer of its context. When the scratch buffer is exhausted, the BPF function will stop resubmitting requests and return to user space immediately. After WiredTiger adds those pages into its cache, it will call `read_xrp` again to continue the lookup starting at the previous page. Since we set the size of the scratch buffer to 4 KB, a BPF function can store up to 6 traversed 512 B pages in the scratch buffer, which leaves room for necessary metadata such as the search key.

Interface modifications. To integrate WiredTiger with XRP, we replace normal read calls with `read_xrp`. `read_xrp` is called when the next page is not in the cache and needs to be read from disk. The eviction policy of WiredTiger enforces that only the pages without any cached children pages can be evicted, so any uncached page will not have cached descendants. Therefore, it is safe to call `read_xrp` to read all of the remaining path from disk without checking the application-level cache again. If `read_xrp` fails for any reason, WiredTiger falls back to the normal lookup path. We allocate a data and scratch buffer for each WiredTiger session to avoid the overhead of allocating and freeing buffers for every request. WiredTiger sessions synchronously process

Average Lookup Latency (µs)				
# Ops	SPDK	io_uring	`read()`	XRP
1	5.2	13.6	13.4	10.7
2	7.8	20.2	20.6	14.2
3	11.2	28.0	27.4	18.0
4	14.3	35.0	34.0	21.7
5	17.2	42.4	41.5	25.4
6	20.2	49.3	48.8	29.3

Table 3: Average latency of a random key lookup with BPF-KV as a function of the depth of the B^+-tree stored on-disk. # ops is the number of index I/Os per lookup.

one request at a time, which avoids concurrency issues.

6 Evaluation

In this section we seek to answer the following questions:

1. What are the overheads of using BPF for storage (§6.1)?
2. How does XRP scale to multiple threads (§6.2)?
3. What types of operations can XRP support (§6.3)?
4. Can XRP accelerate a real-world key-value store (§6.4)?

Experimental setup. All experiments are conducted on a 6-core i5-8500 3 GHz server with 16 GB of memory, using Ubuntu 20.04, and Linux 5.12.0 with an Intel Optane 5800X prototype. All experiments use O_DIRECT, turn off hyper-threading, disable processor C-states and turbo boost, use the maximum performance governor, and enable KPTI [30]. We use WiredTiger 4.4.0 in the experiments.

Baselines. We compare the following configurations: (a) XRP, (b) SPDK (a popular kernel-bypass library), (c) standard `read()` system calls, and (d) standard io_uring system calls.

6.1 BPF-KV

Latency. To answer the first evaluation question, we measure the performance of BPF-KV on a benchmark that performs a million read operations with keys drawn randomly with uniform probability. The experiment varies the number of levels of the tree that are stored on-disk. In this subsection, we disable caching of data objects and index nodes to focus on the overhead of looking up on-disk items. The measured average latency is shown in Table 3. The leftmost column represents the number of chained I/Os that are required to lookup the key in the index (not including the final data lookup). For example, if the number of operations is 4, then BPF-KV is configured with an on-disk tree of depth 4, and it also needs to issue one more I/O to fetch the key-value pair from the log.

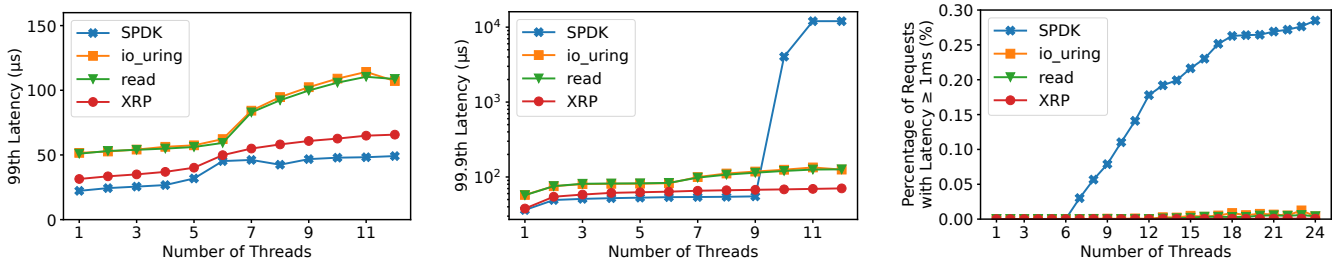
There are a few takeaways from this experiment. First, XRP improves latency over `read()`, because XRP saves one or more storage layer traversals when it traverses the index or moves from the index to the log. Indeed, one can see that XRP's latency increases by about 3.5-3.9 µs for each additional I/O operation, which is close to the device's latency (Table 1). This means that XRP achieves close to optimal latency for resubmitted requests. The same is true for io_uring: in the case of submitting I/O requests synchronously without batching, `read()` and io_uring are almost equivalent. Second, SPDK exhibits better latency than XRP since XRP must pass through the kernel's storage stack once to initiate the index traversal, while SPDK completely bypasses the kernel. Nonetheless, XRP's marginal added latency when the depth of the B^+-tree is increased is close to SPDK's (2.6 µs-3.4 µs). For this reason, in the case of a 6-level index, XRP is only 45% slower than SPDK while `read()` is 142% slower than SPDK. Importantly, XRP achieves this without resorting to polling. This means that, unlike with SPDK, processes can continue to use CPU cores efficiently for other work; XRP's use of CPU time is limited to what is specifically needed to resubmit I/Os in the background and to keep I/O device utilization high.

Figure 5a and Figure 5b present the 99th-percentile latency and 99.9th-percentile latency of XRP, respectively. When running with a single thread, similar to the average latency results, XRP reduces both 99th-percentile latency and 99.9th-percentile latency by up to 30% compared to `read()` and io_uring. Note that our experiment runs as a closed loop, so XRP is running at a higher throughput than `read()` and io_uring. At identical throughput XRP would show additional improvement over these baselines. Interestingly, when the number of threads exceeds the number of cores (6) by more than 3, SPDK's 99.9th-percentile latency increases significantly. This is due to the fact that with SPDK all threads are busy-polling, and cannot effectively share the same core with other threads. To this end, we measure the percentage of requests whose latencies are greater than or equal to 1 ms and present the data in Figure 5c. The results show that SPDK has 0.03% of such requests with 7 threads, and this percentage increases to 0.28% when the number of threads reaches 24. In contrast, io_uring, `read()`, and XRP always have fewer than 0.01% of such requests.

Throughput. Figure 6a shows the throughput of XRP. As expected, as the index depth increases, XRP's speedup is higher compared to standard system calls. Figures 6b and 6c show the throughput speedups with a varying number of threads with an index of depth 3 and 6, respectively. Both figures show the speedup of XRP relative to issuing standard system calls does not decrease even as I/O and XRP BPF functions are scaled across several cores. Once again, XRP provides equal to or higher throughput compared to SPDK once the number of threads is 9 or higher.

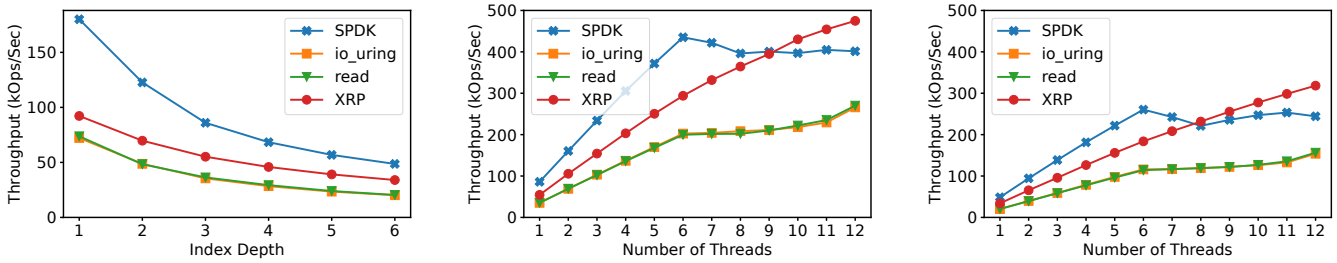
6.2 Thread Scaling

Since storage applications often use a large number of concurrent threads that access I/O devices, for example in order to process concurrent requests and to perform background garbage collection [12, 20, 27, 44], XRP needs to be able to provide good tail latency and throughput under a large number threads. We analyze how XRP scales as a function of the num-



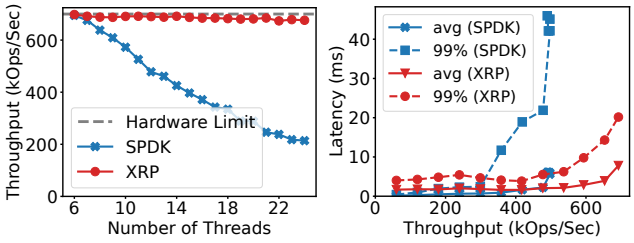
(a) 99th-percentile latency. **(b)** (Log scale) 99.9th-percentile latency. **(c)** Percentage of requests with latency $\geq$ 1 ms.

Figure 5: Tail latency and percentage of requests with extreme latency of XRP and SPDK against `read` and io_uring with BPF-KV with index depth 6, random key lookups, and closed-loop load generator.



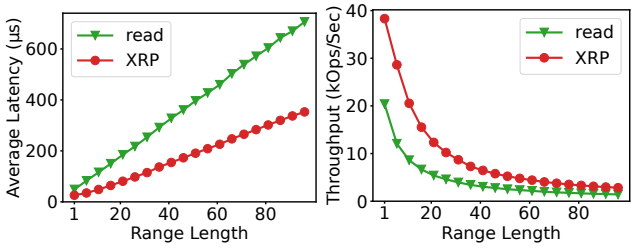
(a) Single thread with varying index depth. **(b)** Multiple threads with index depth 3. **(c)** Multiple threads with index depth 6.

Figure 6: Throughput of XRP and SPDK against `read` and io_uring with BPF-KV with random key lookups and closed-loop load generator.



(a) Scalability of XRP and SPDK. **(b)** Latency-throughput graph of XRP and SPDK with 12 threads.

Figure 7: XRP vs. SPDK with open-loop load generator.



(a) Average Latency **(b)** Throughput

Figure 8: Average read latency and throughput of BPF-KV with XRP vs. `read()` when performing a range query over a varying number of objects.

ber of threads and compare it to SPDK. We run an open loop experiment, where the amount of load matches the maximum bandwidth of the Intel device (5M IOPS for 512 B random reads). Figure 7a compares the throughput of XRP (integrated with io_uring) to SPDK with BPF-KV using 6 on-disk index levels, where each thread represents a different tenant. Two major observations are: 1) when using 6 working threads (the number of CPU cores on the machine) both SPDK and XRP can achieve a throughput close to the hardware limit (the grey dashed line); 2) once the thread count exceeds the CPU cores, SPDK's throughput steadily decreases while XRP still provides stable throughput. SPDK's throughput collapse stems from its polling-based approach; SPDK threads never yield, leaving scheduling up to Linux's CFS which works in coarse 6 ms timeslices. However, idle XRP threads will voluntarily yield the CPU to busy threads, so more CPU cycles are spent on actual work. Figure 7b presents the throughput-latency relationship under 12 working threads as a function of the load. With more threads than CPU cores, both average and tail latencies also increase more significantly in SPDK, as each thread waits longer to be scheduled than in XRP.

6.3 Range Query

Figure 8 compares the average latency and the throughput of running a range query with XRP against performing the query with `read()` system calls. In both cases the range query performs a single index traversal to find the first object, and traverses the leaf nodes of the index to find the address of subsequent objects. The index depth is 6 in this experiment. Even though the XRP range query can only retrieve 32 objects per syscall, the results show this adds negligible overhead. XRP's performance speedup remains relatively constant as a function of the length of the aggregation, since XRP performs only one storage stack traversal for every 32 values retrieved.

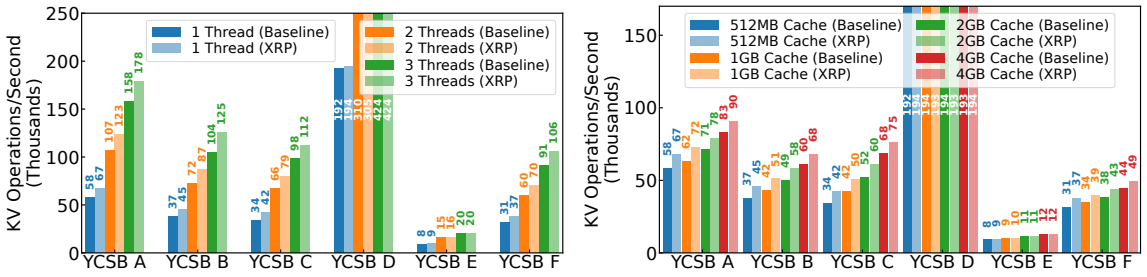


Figure 9: Throughput of reads/scans in WiredTiger with **(a)** varying client threads with a 512 MB cache and **(b)** varying cache size.

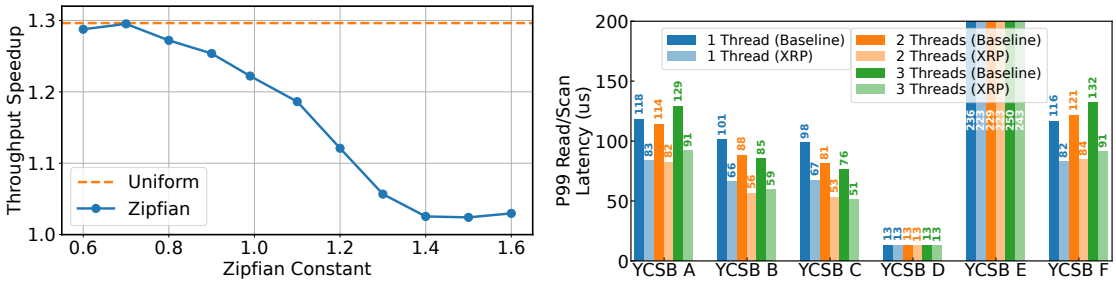


Figure 10: **(a)** Throughput speedup of WiredTiger on YCSB C with a varying Zipfian constant and with a uniform distribution. **(b)** 99th-percentile latency of reads/scans in WiredTiger with varying number of threads with 512 MB cache.

6.4 WiredTiger

To understand whether XRP can benefit a real-world database, we evaluate the performance of WiredTiger with and without XRP on YCSB [41]. We run the different YCSB workloads so that their runtime takes more or less the same time: YCSB A, B, C and E use 10M operations, D uses 50M operations and E uses 3M. The baseline WiredTiger uses `pread()` to read B-tree pages, while the WiredTiger with XRP uses `read_xrp()`. We populate the database with 1 billion key-value pairs and set the size of both key and value to 16 B. The total size of the database is 46 GB. WiredTiger runs eviction threads to evict pages when its cache usage is close to full, and we set the number of eviction threads to 2.

Throughput. Figure 9 shows the total throughput of WiredTiger with different cache sizes and different numbers of client threads. We configure WiredTiger with 512 MB, 1 GB, 2 GB, and 4 GB cache sizes to ensure that WiredTiger can cache at least 1% of its database while not exhausting all the available memory on the machine. We run up to 3 client threads to avoid context switches. The results show that XRP speeds up most workloads consistently by up to 1.25×. The throughput improvements are mostly affected by the cache size. The speedup generally goes down when the cache size becomes larger. In general, XRP provides a lower speedup on WiredTiger than on BPF-KV, because WiredTiger is less optimized than BPF-KV for reading from fast NVM storage, and only spends 63% of its total time on I/O. In particular, XRP does not provide significant improvements on YCSB D and YCSB E. This is because YCSB D follows a latest distribution where the newly inserted items are the most popular ones. Since new inserts are always written into in-memory buffers, most read operations read from those buffers in YCSB D. On the other hand, YCSB E only has inserts and scans. WiredTiger supports scans via an iterator interface, which only looks up one key-value pair at a time. XRP can only benefit the lookup of the first key-value pair of a scan operation, since the rest of the key-value pairs mostly either reside on the same leaf node or require only one additional I/O to fetch the next leaf node.

To study the effect of access distribution on XRP, we run YCSB C with a varying Zipfian constant and with a uniform distribution. Figure 10a shows that XRP's benefit decreases when the Zipfian constant becomes larger (i.e. , the distribution is more skewed) because of the increased cache hit ratio. Note that skews greater than 0.99 represent very high skew levels. We also see that the throughput gain on WiredTiger is lower than that on BPF-KV with the uniform YCSB C. This is again because WiredTiger spends 37% of its total time on non-I/O operations.

Tail latency. We measure the tail read latency of WiredTiger with and without XRP under a fixed load: 20 kop/s per client thread for YCSB A, B, C, D, F, and 5 kops/s per client thread for YCSB E. Since YCSB E has scans instead of reads, we set a lower load for it and measure the tail scan latency instead of the tail read latency. Figure 10b shows that XRP can reduce the 99th-percentile latency by up to 40%. Similar to the throughput, the 99th-percentile latency improve-

ment mostly decreases with a larger cache size, and XRP does not have significant effect on YCSB D and E.

7 Related Work

There are four areas of related work: (a) using BPF to accelerate I/O (typically networking), (b) kernel-bypass systems, (c) near-storage compute, and (d) extensible operating systems and library file systems.

BPF for I/O. There is a large number of systems and frameworks that use BPF to accelerate I/O processing, primarily focused on networking and tracing use cases [2,4–6,15,18,25,28,37,46,49,50,52]. Most closely related to XRP, XDP [28] accelerates networking I/O by adding a hook in the NIC driver's RX path. It then provides an interface for eBPF programs that either filter, redirect, or bounce the packet.

There are no existing systems that use BPF to resubmit storage requests from within the kernel. Kourtis et al. [62] propose a system that uses eBPF functions as an interface to submit disaggregated storage requests in order to avoid crossing the network. In their system, resubmissions occur from a user space service sitting at the host and are not serviced by the kernel itself, since the network is the primary bottleneck (not the kernel software stack). ExtFUSE [36] allows user space file systems on Linux to load BPF functions into the kernel to serve low-level file system requests and thus eliminates unnecessary context switches. While ExtFUSE accelerates user space file systems, it provides no performance benefits for an application that already uses a standard kernel file system (e.g., ext4), since it does not allow applications to bypass the kernel's storage stack. BMC [49] uses BPF to accelerate memcached by intercepting packets on the network path at the host. The BPF functions can then access a separate small kernel-based cache, which serves as a first-level cache and is not synchronized with the user space memcached application. Zhong et al. [85] provide motivation for using BPF for accelerating storage from within the kernel, but do not provide a concrete design, implementation or evaluation.

Kernel bypass. In order to reduce the kernel's overhead when processing I/O, several libraries and operating systems have been designed to let users directly access I/O devices [7,33,34,42,47,57,65,69,71,72,82–84]. Most relevant to our work, Intel's SPDK [82] is a popular kernel-bypass library for storage. In general, the downside of allowing users to access I/O directly is that applications must directly poll for I/O to obtain high performance. This means that cores cannot be shared among processes, which leads to significant underutilization when I/O is not the bottleneck.

Near-storage compute. There are several systems that allow applications to offload their storage functions to the processor embedded within or attached to a storage device [16, 22, 31, 38, 43, 51, 55, 61, 63, 74, 75, 77, 81]. The downside of this approach is that it requires specialized storage devices, dedicated hardware, or both.

Extensible operating systems and library file systems. Our approach is reminiscent of extensible operating systems and library file systems from the 1990s. Extensible operating systems (e.g., SPIN [35] and VINO [76, 79]) allow extension of kernel functionality via user-defined functions. For example, a client can write kernel extensions that read and decompress video frames from disk. Another related approach is library file systems, such as XN [48,56]. Similar to XRP, XN allows userspace library file systems to load untrusted metadata translation functions into the kernel, while guaranteeing disk block protection without understanding file systems' data structures. These approaches required using dedicated operating and file systems, while XRP is compatible with Linux and its standard file systems. ExtOS [32], a more recent extensible OS, minimizes data movements in `read()` and `splice()` by using BPF functions to filter data before copying them to user space or another file, but it still incurs the full storage stack overhead and does not allow I/O request resubmissions.

8 Conclusions and Future Work

BPF has the potential to accelerate applications using fast NVMe devices by moving computation closer to the device. XRP lets applications write functions that can resubmit dependent storage requests to achieve speedups close to kernel-bypass while retaining the advantages of being OS-integrated. Beyond fast lookups, we envision XRP can be used for many types of functions such as compaction, compression and deduplication. In addition, XRP in the future can be developed as a common interface for other use cases where computation needs to be moved closer to storage, such as programmable storage devices and networked storage systems. For example, XRP could be used as an interface that can dynamically support both in-kernel offloading, as well as offloading functions to a smart storage device or an FPGA. Another direction we plan to explore is networked storage. XRP storage functions could be chained with XDP networking functions to create a datapath that bypasses both the kernel's networking and storage paths.

9 Acknowledgments

We would like to thank our shepherd, Ymir Vigfusson, and the anonymous reviewers for their helpful comments. We also thank Kostis Kaffes and Tom Anderson for providing valuable feedback on earlier versions of our work. This work was supported by NSF grants CNS-2143868, CNS-2104292, and CNS-1750558, and an equipment gift from Intel.

References

[1] 3D Xpoint: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[2] bcc. https://github.com/iovisor/bcc.

[3] bpf: Introduce function-by-function verification. https://lore.kernel.org/bpf/20200109063745.3154913-4-ast@kernel.org/.

[4] bpftrace. https://github.com/iovisor/bpftrace.

[5] Cilium. https://github.com/cilium/cilium.

[6] Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/.

[7] DPDK Data Plane Development Kit. https://www.dpdk.org/.

[8] eBPF. https://ebpf.io/.

[9] Efficient io with io_uring. https://kernel.dk/io_uring.pdf.

[10] HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[11] Intel® Optane™ SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html.

[12] LevelDB. https://github.com/google/leveldb.

[13] libbpf. https://github.com/libbpf/libbpf.

[14] Linux Socket Filtering Documentation. https://www.kernel.org/doc/Documentation/networking/filter.txt.

[15] MAC and Audit policy using eBPF. https://lkml.org/lkml/2020/3/28/479.

[16] NGD systems newport platform. https://www.ngdsystems.com/technology/computational-storage.

[17] NVMe base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4b-2020.09.21-Ratified.pdf.

[18] Open-sourcing katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[19] Optimizing Software for the Next Gen Intel Optane SSD P5800X. https://www.intel.com/content/www/us/en/events/memory-and-storage.html?videoId=6215534787001.

[20] Percona TokuDB. https://www.percona.com/software/mysql-database/percona-tokudb.

[21] pread(2) - Linux manual page. https://man7.org/linux/man-pages/man2/pread.2.html.

[22] SmartSSD computational storage drive. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[23] A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[24] Toshiba memory introduces XL-FLASH storage class memory solution. https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html.

[25] udplb. https://github.com/moolen/udplb.

[26] Ultra-Low Latency with Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf.

[27] WiredTiger storage engine. https://docs.mongodb.com/manual/core/wiredtiger/.

[28] XDP. https://www.iovisor.org/technology/xdp.

[29] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 97–112, 2018.

[30] Nadav Amit, Michael Wei, and Dan Tsafrir. Dealing with (some of) the fallout from meltdown. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–6, 2021.

[31] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 56–61, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Antonio Barbalace, Javier Picorel, and Pramod Bhatotia. Extos: Data-centric extensible os. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 31–39, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[34] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high

throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[35] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, 1995.

[36] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

[37] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020.

[38] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, Santa Clara, CA, February 2020. USENIX Association.

[39] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63, 2020.

[41] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[42] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable solid-state storage in future cloud datacenters. *Communications of the ACM*, 62(6):54–62, 2019.

[44] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in RocksDB. In *CIDR*, volume 3, page 3, 2017.

[45] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[46] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. Partition-aware packet steering using XDP and eBPF for improving application-level parallelism. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, ENCP '19, page 27–33, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[48] Gregory R Ganger and M Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX Annual Technical Conference*, pages 1–17, 1997.

[49] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[50] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[51] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon

Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. *SIGARCH Comput. Archit. News*, 44(3):153–165, jun 2016.

[52] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[53] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOSt: Fast & flexible user-space delegation of Linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

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

[55] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A flexible, high-performance key-value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384. IEEE, 2017.

[56] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 52–65, New York, NY, USA, 1997. Association for Computing Machinery.

[57] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for µsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[58] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based CPU in container environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 313–328, Renton, WA, April 2018. USENIX Association.

[60] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the load balancer without regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 193–207, 2020.

[61] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 219–231. IEEE, 2017.

[62] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. Safe and efficient remote application code execution on disaggregated NVM storage with eBPF. *arXiv preprint arXiv:2002.11528*, 2020.

[63] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: Rapid prototype for flash storage systems. *ACM Transactions on Storage (TOS)*, 16(3):1–35, 2020.

[64] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[65] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[67] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, January 1993. USENIX Association.

[68] J. Mogul, R. Rashid, and M. Accetta. The packer filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 39–51, New York, NY, USA, 1987. Association for Computing Machinery.

[69] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[70] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[71] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[72] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[73] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44. IEEE, 2018.

[74] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, 2019.

[75] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with FPGAs. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–12, 2020.

[76] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 213–227, New York, NY, USA, 1996. Association for Computing Machinery.

[77] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, October 2014. USENIX Association.

[78] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, et al. Schema-agnostic indexing with Azure DocumentDB. *Proceedings of the VLDB Endowment*, 8(12):1668–1679, 2015.

[79] Christopher A Small and Margo I Seltzer. Vino: An integrated platform for operating system and database research. 1994.

[80] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[81] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. Recssd: Near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 717–729, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[83] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 73–80, 2019.

[84] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay

Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[85] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. BPF for storage: An exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 128–135, New York, NY, USA, 2021. Association for Computing Machinery.

A Artifact Appendix

Abstract

We open-source XRP, a high-performance storage data path using Linux eBPF. The artifact includes the implementation of XRP in the Linux kernel and two key-value stores that leverage XRP to significantly improve throughput and latency.

Scope

The artifact allows readers to run all the experiments in §6 and generate Table 3, Figure 5, Figure 6, Figure 7, Figure 8, Figure 9, and Figure 10.

Contents

The artifact provides the following parts.

1. XRP: the implementation of XRP in the Linux kernel v5.12.0.
2. BPF-KV: a simple key-value store that uses XRP to accelerate both point and range lookups.
3. WiredTiger: a modified WiredTiger (based on v4.4.0) that integrates with XRP to speed up index lookups.
4. My-YCSB: an efficient YCSB benchmark written in C++ for WiredTiger.

Test scripts and drawing scripts are also provided for all the experiments and results in §6.

Hosting

The artifact is hosted on the main branch (commit `fae90c5`) of the Github repository https://github.com/xrp-project/XRP.

Requirements

XRP requires a low latency NVMe SSD on which the overhead of the Linux storage stack is significant. We use Intel Optane SSD P5800X in all the experiments. In the test scripts, we assume that the operating system is Ubuntu 20.04, and there are 6 physical CPU cores on the machine. Other configurations may require changing the scripts accordingly.

TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs

Guanyu Feng[1], Huanqi Cao[1], Xiaowei Zhu[2], Bowen Yu[1], Yuanwei Wang[1], Zixuan Ma[1], Shengqi Chen[1], and Wenguang Chen[1]

[1]*Department of Computer Science and Technology & BNRist, Tsinghua University,* [2]*Ant Group*

Abstract

Out-of-core systems rely on high-performance cache subsystems to reduce the number of I/O operations. While the page cache in modern operating systems enables transparent access to memory and storage devices, it suffers from efficiency and scalability issues on cache misses, forcing out-of-core systems to design and implement their own cache components, which is a non-trivial task.

This study proposes TriCache, a cache mechanism that enables in-memory programs to efficiently process out-of-core datasets without requiring any code rewrite. It provides a virtual memory interface on top of the conventional block interface to simultaneously achieve user transparency and sufficient out-of-core performance. A multi-level block cache design is proposed to address the challenge of per-access address translations required by a memory interface. It can exploit spatial and temporal localities in memory or storage accesses to render storage-to-memory address translation and page-level concurrency control adequately efficient for the virtual-memory interface.

Our evaluation shows that in-memory systems operating on top of TriCache can outperform Linux OS page cache by more than one order of magnitude, and can deliver performance comparable to or even better than that of corresponding counterparts designed specifically for out-of-core scenarios.

1 Introduction

NVMe [45] Solid State Drives (NVMe SSDs) have drawn a wide range of interest because of their high I/O performance. The U.2 interface [48] and PCIe 4.0 standard [47] have also increased the storage density of NVMe SSDs in recent years. For instance, a dual-socket commodity server can mount an array of more than 16 NVMe SSDs to provide tens of TB of storage capacity, tens of millions of random IOPS, and dozens of GB/s of bandwidth while being 20–40 times cheaper than Dynamic Random Access Memory (DRAM).

Although NVMe SSD arrays can improve the aggregated performance and capacity of the system, they still suffer from block-wise I/O accesses and have latencies at least 100 times longer than those of DRAM. To efficiently process datasets that are significantly larger than available memory, out-of-core systems rely on cache sub-systems to maintain frequently operated data in memory. I/O operations can be merged or skipped on cache hits, bridging the performance gap between DRAM and SSDs.

Page cache [46] is a cache sub-system in modern operating systems (OS) that manages data on the granularity of pages (typically 4KB) across DRAM and SSDs. It enables in-memory applications to support out-of-core processing on SSDs without requiring any rewrite through swapping [44] or memory-mapping [43] based on virtual memory.

However, current implementations of page cache encounter issues related to scalability and performance on cache misses owing to global locking on internal data structures [29]. Recent literature [27, 34, 55] indicates that the heavy I/O stack, page faults, and context switching overheads also limit kernel swapping and I/O performance on fast storage devices such as NVMe SSD arrays.

Therefore, data-intensive applications such as databases and data processing systems [6, 12, 15, 20–22, 52, 54] usually design and implement their own user-space block caches (also known as buffer managers) that manage data by blocks (typically of a fixed size that is a multiple of the physical sector size). In contrast to OS page cache, block cache reduces context switching overhead by running mainly in the user space, and supports customization in terms of tuning block sizes and replacement policies to further improve performance.

Nevertheless, designing and implementing block caches and upper-level components imposes expensive development costs. Existing block caches in the user space usually ask users to explicitly acquire/release blocks [12, 20] or manipulate data through an asynchronous interface [54]. Developers often have to re-design and re-implement the entire system according to the API requirements of the block cache, which is non-trivial. To fill the gap between out-of-core performance and development costs, we investigate a new general cache mechanism that can transparently extend in-memory systems

for efficient out-of-core processing on NVMe SSDs without requiring any manual modification.

Efficient kernel-bypass I/O stacks, such as SPDK [50], can achieve good out-of-core performance in the user space by avoiding expensive kernel I/O operations to take advantage of the high IOPS from the NVMe SSD array. It inspires us to explore a user-space solution that can eliminate the overhead due to page faults and context switching. A solution in the user space is cross-platform, easy to deploy and customize, and avoids introducing potential security vulnerabilities caused by kernel modifications.

To ensure transparency for the user, a *virtual memory interface* is expected to fill the semantic gap between fine-grained memory accesses in existing in-memory programs and block-wise I/O operations on physical block devices as in the case of the virtual memory provided by OS. A virtual memory interface makes it possible for in-memory software to run on NVMe SSDs without requiring any modification if we can automatically redirect memory accesses to the block cache.

Several challenges need to be addressed to implement a user-space block cache with a virtual memory interface. First, the cache system requires good scalability to achieve high out-of-core performance so that it can fit in the hundreds of CPU cores and the tens of millions of SSD IOPS in use today. Second, it requires an efficient address translation mechanism that looks up in-memory addresses for cached blocks, to provide a virtual memory interface with fine-grained accesses. Such fine-grained accesses and per-access address translations pose a much more significant challenge than block lookups in current user-space block caches. Third, it requires a scheme to redirect the memory accesses of existing in-memory systems to the cache system without any manual modification.

To address the above challenges, we propose the following contributions:

- We build a scalable block cache based on a concurrency mechanism named *Hybrid Lock-free Delegation* that combines message passing based delegation with lock-free hash tables. It can utilize the NVMe SSD array with only a few server threads.
- We design a two-level *Software Address Translation Cache* (SATC) to support lightning-fast address translation in the user space, replacing human effort for writing block-aware code with an automatic mechanism by exploiting locality at runtime. SATC can accelerate software address translation by some orders of magnitude.
- We propose a pure software-based scheme to supervise memory accesses based on *compile-time instrumentation* and library hooking techniques. Existing in-memory applications can efficiently run on NVMe SSDs through the block cache without requiring any code modification.

Based on these techniques, we design and implement a user-transparent block cache providing a virtual memory interface, named TriCache. Our results show that TriCache enables in-memory programs to efficiently process out-of-core datasets without requiring manual code rewrite, by using various domains of application. TriCache can outperform OS page cache by some orders of magnitude, and can often reach or even exceed the performance of specialized out-of-core systems.

2 Background and Motivation

In this section, we briefly introduce the two types of general caches that can be used for out-of-core processing, OS page cache and user-space block cache, and use a motivating example to show the benefits as well as the challenges of a new approach that combines the advantages of both.

Page cache is a transparent cache for pages originating from storage devices [46]. Modern operating systems keep the page cache in unused portions of the main memory. Some accesses to storage devices can be handled by the page cache to improve performance. The page cache is implemented in kernels through virtual memory management and is mostly transparent to applications. Users can use a memory-mapping system call [43] to map a file to a segment in virtual memory, or rely on swapping [44] to swap out/in pages to/from disks on-demand, thus accessing storage just like memory.

While the memory interface of the page cache provides maximal user transparency for developing out-of-core applications [9, 26], its use can lead to severe performance bottlenecks, especially on cache misses when the backed storage is an array of high-performance NVMe SSDs. It results from various factors, including but not limited to its global locking in the kernel, the heavy I/O stack, page faults, and context switching overheads [27, 29, 34, 55]. Although some studies have attempted to modify the kernel to improve the performance of the page cache [27–29, 39, 41], it is challenging to apply the relevant modifications in the kernel space, which may introduce potential portability and security issues.

To this end, most out-of-core systems design and implement their own block caching components in the user space to mitigate and even eliminate the above issues. Like the OS page cache, a block cache manages a pool of pages in memory, and loads/evicts pages from/to disks upon user requests. The major difference is that the block cache runs mostly in user space and provides a block interface. Users first `pin` the blocks to be accessed in memory, then read/write data in corresponding blocks, and finally invoke `unpin` to mark the blocks that can be evicted or flushed to storage later, when needed according to the replacement policy [15, 20]. There are also some other forms of the block interface, such as asynchronous `read`/`write` routines with user-defined callbacks [54]. Block cache may be further customized for better performance according to the needs of the application. For example, it is unnecessary to support writing blocks back to the storage if cached contents are known to be read-only [12].

While an efficient and scalable block cache can make full use of storage devices in terms of performance, its block interface requires a considerable amount of work to be put

```
size_t strlen_memory(char* str) {
  size_t len = 0;
  while (str[len] != '\0')
    ++len;
  return len;
}
```

```
size_t strlen_block(str_in_block s) {
  size_t len = 0;
  size_t block_id = get_block_id(s);
  size_t block_off = get_block_offset(s);
  char* raw_ptr = pin(block_id);
  while (raw_ptr[block_off] != '\0') {
    len += 1;
    block_off += 1;
    if (block_off == BLOCK_SIZE) {
      unpin(block_id);
      block_id += 1;
      block_offset = 0;
      raw_ptr = pin(block_id);
    }
  }
  unpin(block_id);
  return len;
}
```

Figure 1: Out-of-core implementations of `strlen` with memory (upper) and block (lower) interfaces

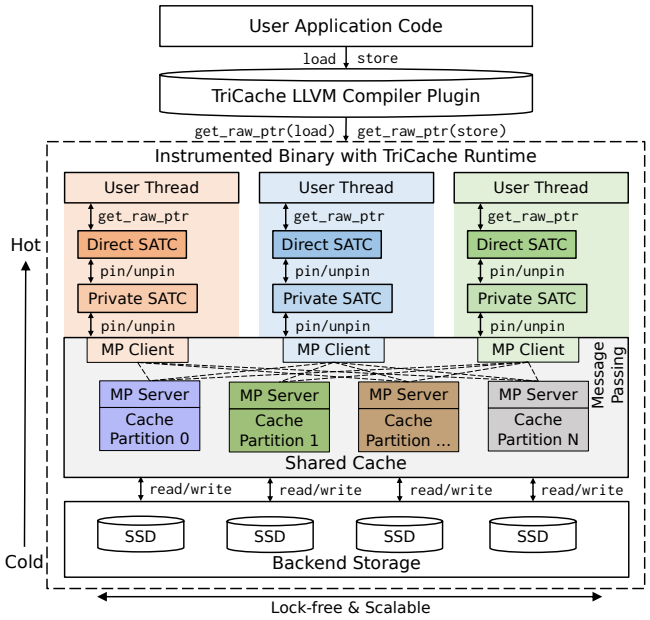


Figure 2: High-level architecture of TriCache

into use. Figure 1 illustrates this with a concrete example: calculating the length of a string. The upper part presents the implementation by using a memory interface, and the lower part shows an alternative version with a block interface. It is evident that the block version is far more complex than the memory version because system developers have to take care of more details, such as checking the block boundaries and making `pin`/`unpin` calls manually, while the memory version only needs to perform memory accesses.

It thus motivates us to explore a block cache providing a virtual memory interface in the user space, that can combine the advantages of high out-of-core performance and high user transparency from both types of caches. The user-space approach drops some functional capabilities of the OS page cache, such as sharing memory across processes with consistency guarantees. However, it allows us to redesign the cache sub-system towards new high-performance storage. Although the virtual memory interface forces applications to manipulate the cache synchronously and manage data in fixed-size blocks (rather than objects or rows), such an interface enables user transparency and saves developers considerable effort.

However, a user-space block cache with a virtual memory interface is not as easy as it might appear. Since every memory access now needs to involve a pair of `pin` and `unpin` calls to ensure that the data accessed reside in memory, as well as given that `pin` and `unpin` imply storage-to-memory address translation and concurrency control operations[1], we need optimizations in addition to those in current block cache designs to make `pin`/`unpin` as fast as possible.

[1] In case of cache misses, the victim blocks resident in memory need to be replaced with the requested blocks on storage; in case of cache hits, the reference counts need to be updated with locks/latches or atomic operations.

3 Design and Implementation of TriCache

In this section, we first present an overview of the system design of TriCache, and then describe its efficient multi-level block cache runtime in a bottom-up manner, including how to build a scalable block cache and reduce the cost of cache accesses in the user space to support transparent usage. Finally, we introduce how to automatically apply TriCache to in-memory applications via compiler techniques.

3.1 Overview of TriCache

Figure 2 shows the high-level architecture of TriCache. It consists of an LLVM compiler plugin and a runtime module.

TriCache LLVM Compiler Plugin first instruments each memory instruction, such as load and store, in the user application code, inserting a software address translation call (named `get_raw_ptr`) before the memory instructions. Upon execution, the instrumented binary calls the interface every time it tries accessing a storage address and retrieves a memory address pointing to data cached in memory. The translated address is then used as usual for the memory instruction.

TriCache Runtime is the core of TriCache (the dashed box in Figure 2). It is a multi-level block cache that supports fast address translation and provides a virtual memory interface. It implements `get_raw_ptr` to translate blocks to their corresponding cached memory addresses, manages the in-memory data cache for recently accessed blocks, handles I/O operations when the cache misses, and evicts blocks when the cache is full.

In the implementation of `get_raw_ptr`, TriCache Runtime introduces a two-level *Software Address Translation Cache*

(SATC) on top of the conventional block cache (*Shared Cache* in Figure 2). The first level is a directly mapped Direct SATC, and the second level is a set-associative Private SATC (under the three *User Threads* in Figure 2). They serve purposes similar to those of the hardware TLB, and accelerate address translations for hot blocks. We implement them as thread-local metadata caches for storage-to-memory address mappings. Direct SATC is responsible for efficient translation when operating the most recently used entries, while Private SATC aims to provide sufficient entry caching capacity and merge inter-thread operations. Meanwhile, the SATC employs a `pin`/`unpin` protocol (as mentioned in Section 2) to implement an inclusive two-level metadata cache. TriCache deploys SATC to automatically exploit localities in running programs for address translation and reduce the cost of runtime API calls, rather than relying on manually programming against blocks to reduce the number of API calls and amortize the runtime overheads.

Below SATC, TriCache Runtime manages data with Shared Cache (in the middle of Figure 2, gray background). Shared Cache is a full-featured block cache shared by multiple threads that maintains an in-memory cache pool for reading and writing the underlying storage. It manages a block table for all in-memory blocks and serves address translations when SATC misses. The block table exposes a `pin`/`unpin` interface to SATC as well, with the guarantee that recently used data pinned by SATC are not swapped out to external storage. To prevent scaling bottlenecks introduced by locking, the block space is partitioned, and each partition is owned by a single thread. Message passing based delegation is used to render critical operations (including block replacements and I/O accesses) single-threaded and lock-free. Moreover, Shared Cache can use kernel-bypass I/O stacks to eliminate context switching for I/O operations.

For Shared Cache, we propose a Hybrid Lock-free Delegation based concurrency control scheme. First, we distinguish between address translations and data accesses. Only address translations call `pin`/`unpin` remotely through message passing, while data accesses directly manipulate memory and rely on the CPU cache to ensure data consistency. The cached data are thus stored only in the Shared Cache and directly accessed by threads without any redundant memory copies. Second, we design and implement the per-partition block table as a concurrent lock-free hash table to further reduce inter-thread message passes. With this concurrent block table, only pinning operations that are missed in Shared Cache require a synchronous remote call.

In Figure 3, we present an example of a user program, a follow-up of Figure 1, instrumented by and then running with TriCache. The C program is first compiled to LLVM IR (Intermediate Representation) with Clang, with the memory read compiled to a load instruction. TriCache LLVM Compiler Plugin instruments the load instruction into two operations: one calls `get_raw_ptr` to retrieve the translated memory address, and the other loads the cached data. Upon execution, the `get_raw_ptr` call of TriCache Runtime results in an address translation operation sequence. If Direct SATC hits, the result is returned; otherwise, it pins the corresponding block in Private SATC. If the block is found in Private SATC, the result is returned; otherwise, it pins the corresponding block in Shared Cache. If Shared Cache is holding the block, the pin operation finds the memory address of the cached block in the concurrent block table. Otherwise, as invoked by a remote call, Shared Cache reads the block from storage and loads it into memory.

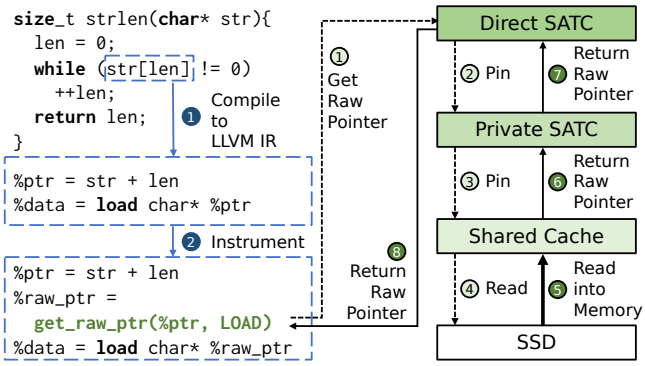


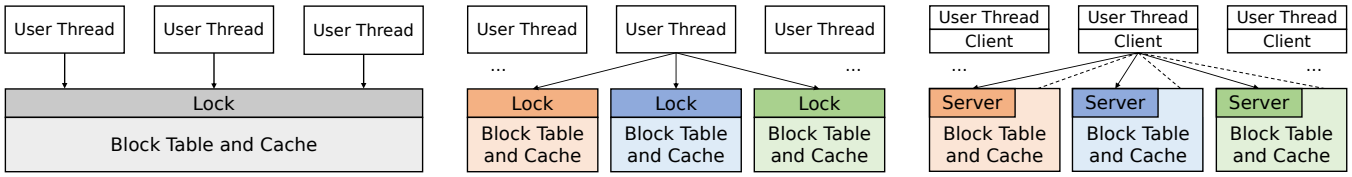
Figure 3: An example of a user program running on TriCache

3.2 Shared Cache

As the core module of TriCache, Shared Cache determines TriCache's throughput, especially its I/O performance. Therefore, good scalability is the primary design goal of Shared Cache for the effective use of hundreds of CPU cores, tens of NVMe SSDs, and millions of IOPS.

Design Decisions. Figure 4a shows a straightforward design used by the current Linux Kernel. It uses a global lock to protect the block table (or page table) and the cache. However, the single lock leads to heavy lock contention and is difficult to scale for high-performance storage devices [29]. The sharding technique can help mitigate the scalability issue, as shown in Figure 4b. The block cache [12, 29] can use a predefined function (usually hashing) to partition the blocks into several shards and then assign a lock to each shard. In addition, recent work proposes that well-designed delegation based on message passing can provide better scalability and hotspot tolerance than locks [23, 33, 53] on NUMA (Non-uniform Memory Access) architectures.

Therefore, we propose a Hybrid Lock-free Delegation for Shared Cache of TriCache, as shown in Figure 4c. The Shared Cache adopts a client-server model based on message passing (solid lines in Figure 4c). Each client-server pair shares a lightweight message queue with a size of two cache lines, similar to ffwd [33]. Each user thread corresponds to a client, and several dedicated servers handle requests from clients.



(a) Global Locking (b) Sharded Locking (c) Hybrid Lock-free Delegation

Figure 4: Different designs and concurrent mechanisms for the shared block cache

Each server is single-threaded, lock-free, and only responsible for managing a part of the blocks (e.g., partitioned by hashing block IDs). Multiple partitions and servers can achieve concurrency and scalability, and more servers can be added when a higher throughput is desired. In addition to message passing based delegation, clients can directly access per-partition block tables on cache hits to reduce server-side CPU consumptions (dashed lines in Figure 4c; more details are provided in the *Client-side Fast Paths on Cache Hits* paragraph below).

Metadata-only Delegation. When a user thread accesses block data, TriCache divides the block access into a metadata operation and a data operation. Metadata operations include address translations, reference count management, and evict policy enforcement. Data operations are memory accesses, such as load and store. In TriCache, only metadata operations are processed by servers through delegation while clients issue data operations by themselves.

A block is accessed in three stages. In the first stage, the client asks the server to cache the block in memory (`pin`) and translate the storage address to its address in memory. The server updates the metadata of the requested block, reads uncached blocks, and evicts unused blocks, without touching the actual data. In the second stage, after receiving the response, the client will directly perform its memory access on the translated memory address. In the last step, the client notifies the server that the block has been released and can be further evicted by the server (`unpin`).

This design eliminates redundant memory copies between servers and clients. Servers focus on metadata operations so that a few server threads can achieve good performance. Meanwhile, it helps TriCache provide the same consistency and atomicity guarantees as memory, which is necessary for user transparency and compatible with in-memory applications. CPU directly executes data operations on the client side via memory instructions, and cache coherence is ensured by hardware. TriCache only needs a memory fence to ensure that the updates are visible before evicting modified blocks.

Client-side Fast Paths on Cache Hits. We propose using concurrent block tables to avoid server-side synchronizations on cache hits. A client first tries to directly find a block in the block table and update the block reference counts (number of clients in use) by using atomic operations. If it succeeds,

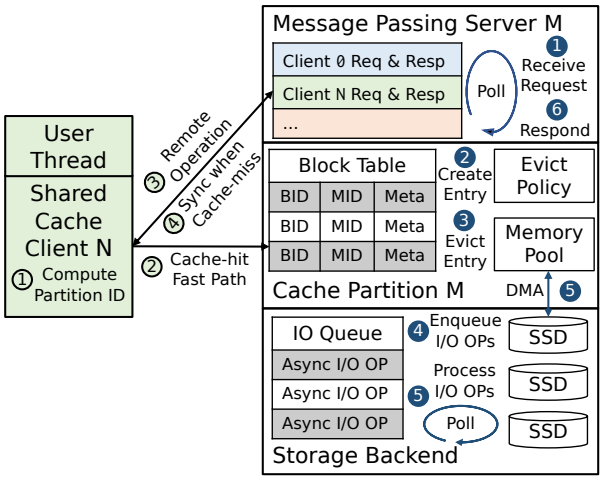


Figure 5: Shared Cache of TriCache

the client can translate the address from the concurrent block table by itself, thus skipping synchronous message passing. The client then sends an asynchronous message if this direct operation changes the reference count from 0 to 1 or conversely, to notify the server to update the evict policy for the block. Multiple asynchronous messages can be batched and processed together to amortize message passing overheads.

Workflow and Implementation. Figure 5 shows the workflow and implementation of Shared Cache. Each user thread corresponds to a Shared Cache Client and gets its unique Client ID. Each partition has a polling-based message passing server that is used to process requests sent from clients and return results to them. In addition, each partition maintains a concurrent block table for blocks cached in the memory, and each entry in the block table stores the Block ID (BID) of the block, Memory ID (MID) of its in-memory cache, and its metadata (Meta). The metadata include information on whether the block is available and whether it has been modified, and the reference count pertaining to the clients in use. We use the compact hashed block table similar to [51], in which each entry occupies only an average of 8 bytes, and uncached blocks do not occupy memory.

The cached blocks are indexed by their Memory IDs and stored in a Memory Pool. Meanwhile, an Evict Policy tracks all cached blocks with a zero reference count as they can be

safely evicted. Every time the cache is full, the Evict Policy chooses and evicts one or more blocks based on its statistics and strategies. TriCache uses the CLOCK algorithm [40] by default, and it is replaceable, allowing users to customize the policy based on their application characteristics. In addition, policy implementations in TriCache are completely single-threaded, so users do not need to consider any concurrency issues. At the bottom, an asynchronous I/O backend (Storage Backend) manages pending IO requests in an I/O Queue to continuously poll and process I/O operations. The I/O backend is also customizable and defaults to SPDK that is backed by user-space NVMe drivers. A kernel-space alternative based on Linux AIO is also supported as another candidate.

When a user thread operates on a block, its client (*N* in the figure) first computes the Partition ID (*M* in the figure) by a predefined partition function. The client then searches for a valid block entry from the block table of Partition *M*, and if such an entry exists, the client tries increasing the reference count by using atomic operations. If the atomic operations succeed on cache hits (*Cache-hit Fast Path* in Figure 5), the client pins the block in memory and can directly query the memory address of the cached block. The client may further send an asynchronous request to the server if it is updating the reference count from 0 to 1, or the converse. The server then performs the corresponding actions according to the Evict Policy, such as enabling or disabling evictions of the block.

If the atomic operations fail on cache misses or when the block is being swapped in/out, the client requests a remote operation via synchronous message passing, immediately releases CPU resources, and waits for responses from the server (*Remote Operation* in Figure 5). After receiving the request, the server creates a block table entry and sets its valid bit to `false`. If the block table is full, the server evicts blocks according to the Evict Policy and sets their valid bit to `false`. The server then appends I/O operations for new blocks and the evicted blocks to the I/O queue. The I/O Backend processes the I/O requests by polling and controlling NVMe SSDs to perform DMA operations directly on the Memory Pool. And the server sets valid bits to `true` once the I/O requests have been processed, and it sends the memory addresses of the blocks to clients via message passing. After receiving the response, the client resumes and performs its memory accesses.

We use a micro-benchmark on a 128-core machine to test the effectiveness of TriCache Shared Cache. It can scale linearly to 256 threads (1/8 of the threads are servers), reaching 96.8M ops/s, and the hybrid mechanism provides an improvement of 52% compared with the delegation-only approach.

3.3 Software Address Translation Cache

The Shared Cache of TriCache provides scalable I/O performance and an efficient set-associative cache. However, block table lookups and atomic operations are required for each access on cache hits, still limiting the performance of TriCache.

Guiding Ideas. Considering the manual use of the block cache (e.g., Figure 1), users call the `pin` interface to get the in-memory address for a block, and then use the memory address to perform multiple operations; they finally call the `unpin` interface to release the block. Multiple read and write operations can be performed between a pair of manual `pin` and `unpin` operations to reduce the number of cache lookup operations. Users manually take advantage of data locality while investing extra effort in development.

In contrast, we design TriCache to automatically exploit locality to simulate manual coding without requiring human effort. We propose to build a two-level Software Address Translation Cache (SATC) on top of Shared Cache. The higher-level cache stores hotter data and provides faster access and smaller capacities than the lower-level cache, similar to the multi-level cache of the CPU and hierarchical storage [32, 49, 56]. Based on this idea, we now show how to implement the multi-level cache in software and where to divide the levels.

SATC Design. In our design, only the last-level cache manages data, and higher-level translation caches manage only metadata, such as modifying the reference counts of the blocks and translating block IDs to memory addresses. Managing metadata instead of data can help avoid redundant memory consumption, additional memory copies, and memory consistency issues caused by the multi-level design. The multi-level cache of TriCache is designed to be an inclusive cache, which means that all blocks in the higher-level cache are also present in the lower-level cache. With this inclusive policy, higher-level caches need to only interact with their next level. Moreover, TriCache guarantees that the capacity of higher-level caches is no greater than the lower-level cache, thus eliminating out-of-space errors from the lower-level cache when the higher-level cache requests to swap in blocks.

On top of the Shared Cache, we build a thread-local set-associative cache called Private SATC. When the Private SATC hits, the user thread uses its thread-local block table and evict policy, and only when the Private SATC swaps in/out blocks does the user thread need to operate on the Shared Cache. Private SATC is purposed to reduce Shared Cache operations for the hot data of its thread. Examples include thread-local hot data when each thread computes a segment of data independently, and hot elements shared by all threads when processing skewed data. Private SATC also helps reduce concurrent block table operations with cross-NUMA memory accesses and false sharing, which could take about 3–8 times higher latency than local memory accesses in our evaluation.

We further build a direct mapping cache called Direct SATC on top of the Private SATC to alleviate overheads due to hash table lookups and evict policy maintenance. Direct SATC maintains a few recently accessed pages in a fixed-size array to speed up address translation to a few bitwise operations and avoid having to update the evict policy for each access. The goal of Direct SATC is to cover multiple consecutive

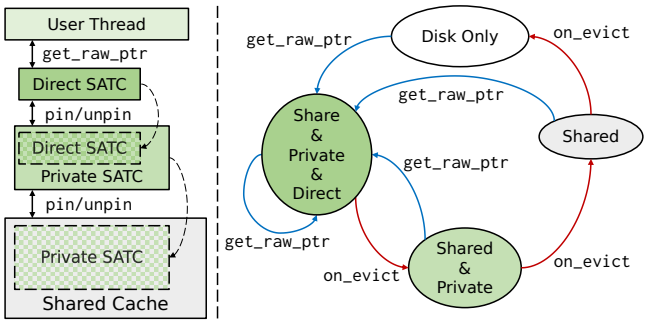


Figure 6: Software Address Translation Cache

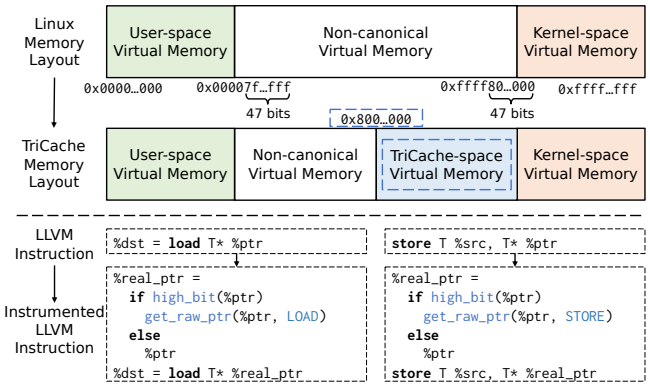


Figure 7: Memory layout and LLVM instrumentation

operations on the hottest blocks, such as sequential reads and writes, and displace manually written `pin`/`unpin` operations.

Implementation. As shown in Figure 6, we implement an inclusive multi-level cache with `pin` and `unpin` interfaces. Blocks in Direct SATC must exist in Private SATC, and blocks in Private SATC should also be present in Shared Cache. Private SATC calls the `pin` interface of Shared Cache to load blocks into the Private SATC, thus increasing reference counts to ensure that Shared Cache does not swap out the blocks. When blocks are evicted from Private SATC, it calls the `unpin` interface of Shared Cache to release the reference count. On top of Private Cache, Direct SATC also uses a scheme similar to Private SATC but provides a single `get_raw_ptr` interface implicitly combining a `pin` call and a following `unpin` call. It implies caching a block and translating the block ID into its raw address in memory. The raw address is valid until the next `get_raw_ptr` call because the subsequent access can evict any previous block from Direct SATC and possibly call the `unpin` interface of Private SATC or Shared Cache.

The right-hand side of Figure 6 presents the state machine maintained in TriCache. Starting from Disk Only state, the user thread loads blocks into Shared Cache, Private SATC, and Direct SATC by calling `get_raw_ptr`. When Direct SATC evicts blocks, Shared Cache and Private SATC still hold them. When the last thread in use evicts a block from its Private SATC, the block enters Shared Cache Only state. Any `get_raw_ptr` re-loads the block into all three levels of caches. If Shared Cache also evicts the block, it is removed from the in-memory cache and written back to storage when it is dirty, ending in Disk Only state.

In our implementation, the aggregated capacity of Private SATC entries is equal to that of Shared Cache entries, and the Direct SATC has a size 1/4 of that of the Private SATC.

Our evaluation shows that SATC can improve performance by tens of times over Shared Cache on real-world workloads. When SATC can absorb all accesses, TriCache can reach 57% and 91% of the in-memory performance for purely random and nearly sequential access patterns respectively, making it practical to operate the block cache at per-access granularity.

3.4 Compile-time Instrumentation

With the help of SATC, TriCache opens up opportunities to provide a virtual memory interface and make TriCache fully transparent to users. To this end, we propose a purely user-space scheme based on compile-time instrumentation and library hooking techniques.

Memory Layout. We first modify the memory layout as shown in the upper part of Figure 7. In Linux, the current x86_64 virtual memory layout (with four-level page tables) consists of three main parts. User-space takes 47 bits at the beginning, kernel-space occupies 47 bits at the end, and most of the space in the middle is a hole of non-canonical virtual memory. We map the TriCache-managed disks into unused holes by block size, starting from 0x800...000 as TriCache-space virtual memory. Memory addresses in TriCache-space can be translated into an actual user-space memory address by calling the `get_raw_ptr` interface of TriCache Direct SATC.

Instrumentation. To enable transparent read and write operations on top of the TriCache block cache, we perform a translation before each memory operation (e.g. load, store, and atomic operations), so that TriCache-space addresses can be used just like user-space memory. TriCache does not require any manual code modifications with the help of compile-time instrumentation. The lower part of Figure 7 shows pseudo-codes for instrumenting load and store instructions in LLVM IR. TriCache instrumentation takes the highest bit of addresses to determine whether a memory address refers to user-space memory or TriCache-space virtual memory.

While instrumentation provides the virtual memory interface for TriCache-space memory, we still need to determine what data should be placed in TriCache-space memory. First, data on the stack is not necessary to enter the block cache. We perform a data flow analysis from LLVM `alloca` instructions to eliminate unnecessary instrumentation and overhead for the stack. Second, we set a runtime threshold for TriCache so that only memory allocations greater than the threshold will

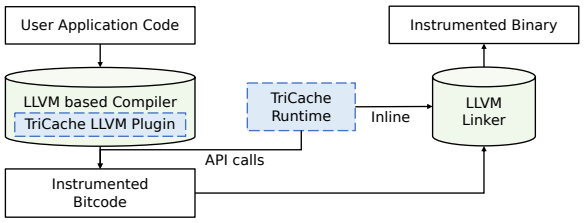


Figure 8: Compiling workflow of TriCache

belong to TriCache-space memory. In contrast, small chunks of data, usually short-term temporary data, remain in-memory allocation. Finally, TriCache supports limiting the total size of data allocated in memory by a predetermined memory quota. If in-memory data exceed the memory quota, TriCache is able to take over later allocations. Users can adjust the above runtime parameters to obtain a balanced trade-off between the memory usage and the performance.

Implementation. Figure 8 illustrates the compiling workflow of TriCache. The user application code is compiled by a compiler based on LLVM. The TriCache LLVM plugin instruments the code and generates instrumented LLVM IR bitcode. The plugin performs instrumentation after all the optimization passes, so it does not affect the compiler optimizations on applications, such as automatic vectorization. Also, TriCache supports vector instructions since TriCache leaves the CPU to perform memory operations.

Then, the bitcode links with the pre-compiled TriCache runtime (including `get_raw_ptr` implementations). TriCache forces to inline the cache-hitting implementations of Direct SATC and Private SATC through link-time optimization (LTO) to avoid intensive function call overheads.

The TriCache runtime also contains APIs on top of the virtual memory interface for manual optimizations, including `pin` and `unpin`. Optionally, users can optimize some bottlenecks of their applications through these APIs, such as using block-wise accesses and prefetching, while leaving other parts to transparently support out-of-core processing by instrumentation. In the TriCache runtime, some common utility functions, such as `memcpy` and `memset`, are already manually implemented by block-wise `pin` and `unpin` to reduce overheads of per-byte address translation from common components.

4 Evaluation

We set up our experiments on a dual-socket server equipped with two AMD EPYC 7742 CPUs (64 physical cores and 128 hyper-threads per CPU) and 512GB DDR4-3200 main memory. The storage devices are 8 PCIe-attached Intel P4618 DC SSDs which provide 51.2TB capacity, 9.6M 4KB-read IOPS, and 3.9M 4KB-write IOPS in total. The server runs Debian 11.1 with Linux kernel 5.10 and uses Clang 13.0.1 to compile TriCache and other systems.

In our evaluation, we limit the total available capacity of DRAM by `cgroups` to evaluate out-of-core performance. For TriCache and other systems with block caches, we ensure that the overall memory is less than the expected memory limit by adjusting the cache sizes. The SSDs are configured in SPDK mode for TriCache and as raw blocks for swapping. If the system requires a single filesystem, we construct a software RAID-0 by `mdadm` and use the XFS filesystem. TriCache launches 16 background threads (bound to 8 cores) for Shared Cache and uses 4KB blocks by default. And the total number of threads are searched to maximize performance over powers of two from the number of hardware threads (i.e. 256).

We first evaluate TriCache on four representative domains in terms of end-to-end performance: graph processing (Section 4.1), key-value store (Section 4.2), big-data analytics (Section 4.3), and transactional graph database (Section 4.4).

We then conduct a micro-benchmark, by using a configurable number of threads that issue load/store instructions. We adjust the hit rates and access patterns to explore circumstances in which TriCache outperforms OS page cache and to assess whether the design of TriCache provides a reasonable trade-off between in-memory (i.e. cache hit) and out-of-core (i.e. cache miss) performance (Section 4.5).

Finally, we use a series of breakdown experiments to evaluate the performance-related impact on TriCache, including I/O backends, hit rates and hit latency of SATC, and number of threads (Section 4.6).

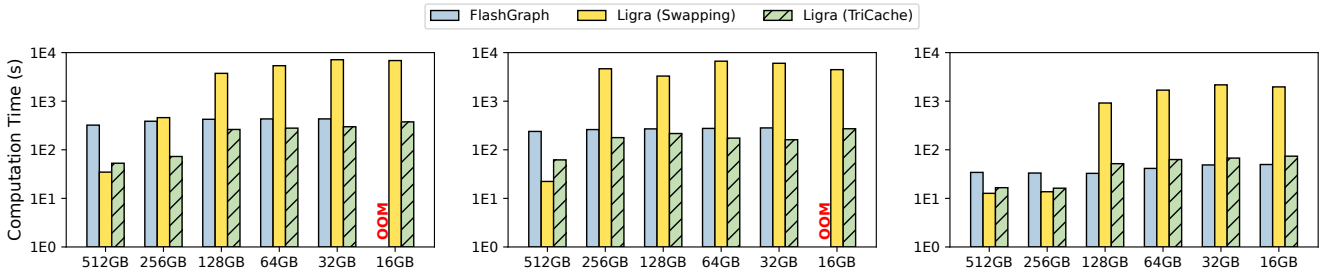
4.1 Performance on Graph Processing

Experimental Setup. Graph processing is a demanding workload for cache systems due to many small and random accesses on large datasets. We transparently apply TriCache to an in-memory graph processing framework Ligra[2] [37] and extend it to out-of-core. The baselines are Ligra with OS swapping and FlashGraph[3] [54], an efficient semi-external memory graph processing framework designed for SSDs.

Both Ligra and FlashGraph use 32-bit vertex IDs, and we force Ligra to use push mode to align with FlashGraph. For FlashGraph, we follow its recommended configuration of creating an XFS filesystem for each SSD block device and binding the device to the corresponding NUMA nodes. Meanwhile, FlashGraph is a semi-external memory graph engine that always stores vertex states in memory and edge lists on SSDs. We thus make TriCache to manage at least edge lists in the cache for a fair comparison.

We evaluate FlashGraph, Ligra on swapping, and Ligra on TriCache by three common graph algorithms: PageRank (PR), Weakly Connected Components (WCC), and Breadth-First Search (BFS). The dataset is a real-world graph dataset, UK-2014 [7, 8], with 788 million vertices and 47.6 billion edges. It requires more than 400GB for Ligra in-memory execution.

[2] https://github.com/jshun/ligra [commit 7755d95]
[3] https://github.com/flashxio/FlashX [commit 2a649ff]



(a) PageRank (b) Weakly Connected Components (c) Breadth-First Search

Figure 9: Computation time of FlashGraph, Ligra on swapping, and Ligra on TriCache (lower is better)

In-memory Performance. Figure 9 shows the computation time of FlashGraph, Ligra on swapping, and Ligra on TriCache under different memory quotas. With 512GB of memory, Ligra can process all three algorithms in memory, and TriCache and FlashGraph can buffer all data in their cache. Under this setting, TriCache incurs overheads of only 34.4% for PageRank, 64.0% for WCC, and 23.5% for BFS. The in-memory performance shows that TriCache can provide efficient address translations and cache hits with its virtual memory interface, owing to the two-level SATC. Meanwhile, TriCache outperforms FlashGraph by 6.08×, 3.85×, and 2.06×, respectively. It illustrates that FlashGraph yields much higher in-memory overheads than TriCache because the block cache of FlashGraph involves redundant memory copies on cache operations with its `read`/`write` interfaces.

Out-of-core Performance. Under 256GB memory limitation, the caches start swapping in/out blocks/pages. Compared to the in-memory performance, Ligra on TriCache saves about half of memory and yields 47.7% performance on PageRank, 12.5% performance on WCC, and 78.4% performance on BFS. And TriCache's speedups over OS swapping and FlashGraph are 6.30× and 5.31× on PageRank, 26.1× and 1.46× on WCC, and 0.85× and 2.05× on BFS, respectively.

As the usable memory further decreases, I/O efficiency becomes the main factor affecting performance. For example, in the case of 64GB of memory, the performance of TriCache is 19.3×, 38.3×, and 26.8× better than that of swapping. Compared with FlashGraph, TriCache can still provide improvements of 54.8% and 58.3% on PageRank and WCC respectively, while the performance of Ligra with TriCache is 34.3% lower than FlashGraph on the BFS algorithm. This is because FlashGraph adopts two-dimension partition for out-of-core graph processing, resulting in a 50.1% cache hit rate that saves 2.68× of I/O volume compared to TriCache. Still, TriCache provides an average I/O bandwidth 1.78× better than FlashGraph and thus reduces the performance gap.

It is noteworthy that the semi-external memory FlashGraph cannot fit vertex states of PageRank and WCC with 16GB of memory. It leads to out-of-memory errors, whereas TriCache can operate the same dataset fully out-of-core.

The above results indicate that TriCache can extend an in-memory graph framework to support out-of-core processing without manual modification and can deliver performance comparable to a well-designed external memory framework. Meanwhile, TriCache outperforms OS swapping by up to 38.3× while providing the same user transparency.

4.2 Performance on Key-Value Stores

Experimental Setup. We use RocksDB[4] [12], a persistent key-value store widely used in production systems, for evaluation in this part. RocksDB organizes on-disk data in immutable Sorted Sequence Tables (SSTs). It provides a block-based table format on top of its user-space block cache, and a plain table format optimized for in-memory performance via `mmap`. We use TriCache to buffer RocksDB plain tables without manual modification and compare it with plain tables based on OS memory-mapped files and block-based tables on RocksDB's own cache.

We use the mixgraph [10] (prefix-dist) workload proposed by Facebook, which models production use cases at Facebook and emulates real-world workloads of key-value stores with hotness distribution and temporal patterns. The keys and values are 48 and 43 bytes on average, respectively, and there are 83% reads, 14% writes, and 3% scans. We generate 2 billion key-value pairs (consuming 180GB of space) and execute 100 million operations. Both plain and block-based tables use the hash index with a 4 bytes prefix. We set the sharding number of the RocksDB block cache to 1024 to avoid lock contentions on our 256-thread server and use the direct I/O mode for the RocksDB block cache. We also disable WAL to prevent log flushing from becoming a performance bottleneck.

In-memory Performance. Figure 10 illustrates the throughput of Plain Tables on TriCache, Plain Tables on `mmap` and Block-based Tables on the RocksDB user-space cache with different memory quotas. In memory, RocksDB Plain Tables with `mmap` provides the best performance, which is 4.28M ops/s. TriCache reaches about 53.5% throughput of `mmap`, and 73.7% throughput of the RocksDB block cache.

[4] https://github.com/facebook/rocksdb [tag v6.26.1]

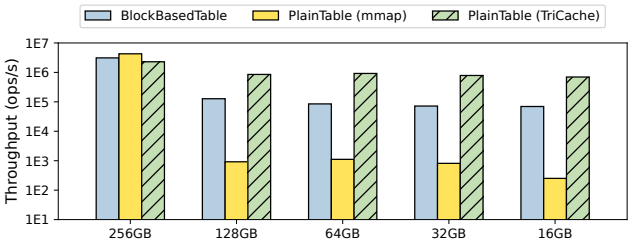


Figure 10: RocksDB throughput with varying memory quotas

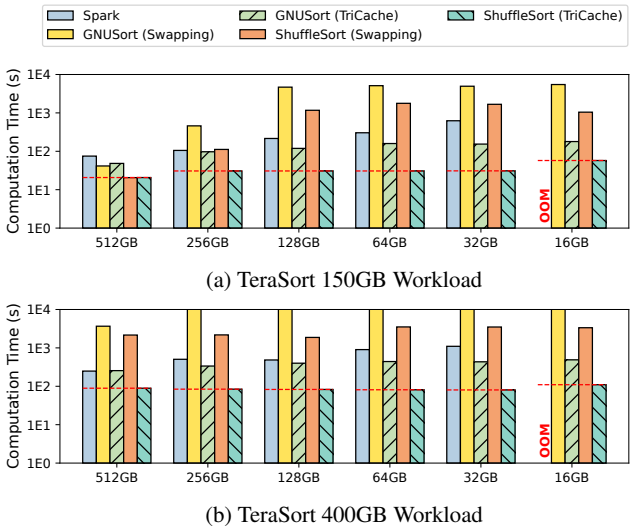
Out-of-core Performance. When RocksDB runs out-of-core, TriCache brings performance improvements of 2–3 orders of magnitude compared with `mmap`. Plain Tables with TriCache outperforms Block-based Tables by 6.69× under 128GB memory, 10.8× with 64GB, 10.9× with 32GB, and 10.0× with 16GB. Some performance benefits of TriCache come from the efficient I/O stack of SPDK, while the excellent scalability of Shared Cache is another key factor. For example, the RocksDB block cache can deliver a throughput of 122K ops/s with 256 threads. However, our eight NVMe SSDs require about 1024 I/O in-flight requests to maximize the I/O performance. Unfortunately, when the number of threads is increased from 256 to 1024, the throughput instead gradually drops. In the case of 1024 threads, RocksDB only provides 71.3% throughput of 256 threads. In contrast, the performance of RocksDB with TriCache improves by 2.15× from 256 threads to 1024 threads.

The in-memory performance indicates that user-transparent TriCache can provide similar performance as manually managed block cache in RocksDB. Meanwhile, TriCache has the potential to help existing systems with in-memory backends, such as RocksDB with Plain Tables, to achieve better out-of-core performance without any manual modifications.

4.3 Performance on Big-Data Analytics

Experimental Setup. TeraSort [1] is a representative application and an important performance indicator in the domain of big-data analytics [16]. Its typical distributed or out-of-core implementation consists of a shuffle phase followed by a sort phase. The shuffle phase produces parallel sequential reads and writes, which is I/O bound [16] and can stress sequential I/O throughput on cache systems. The sort phase requires the cache to buffer the working partition in memory and issues a vast number of string comparisons and copies that can examine the runtime overhead of cache systems.

We generate two TeraSort workloads, 1.5B records (about 150GB) and 4B records (about 400GB). For TriCache, we first use the parallel sort based on multi-way merge sort in GNU `libstdc++` [38] (named GNU Sort) to implement an out-of-core TeraSort, which requires only a single function call. We also implement a shuffle-based parallel sort by partitioning the first byte of the keys (named Shuffle Sort), which takes 15 additional lines of C++ code. Compared with the multi-way merge sort, the shuffle-based parallel sort mainly issues sequential read/write I/O operations, so it is more friendly to out-of-core processing. We use TriCache to manage memory allocations during sorting and compare TriCache with OS swapping. For Shuffle Sort, we configure the page size of TriCache to 128KB to maximize the sequential I/O performance. We also use a widely used big-data framework Spark[5] [52] as a baseline, which supports both scale-up and scale-out processing.



(a) TeraSort 150GB Workload

(b) TeraSort 400GB Workload

Figure 11: Computation time for TeraSort workloads with different memory quotas (lower is better)

In-memory Performance. Figure 11 shows the computation time of TeraSort. On the 150GB dataset, both GNU Sort and Shuffle Sort occupy about 300GB of memory and fit in 512GB of memory. In this case, Shuffle Sort is 2.01× faster than GNU Sort. Meanwhile, the overheads of TriCache amount to only 14% for GNU Sort and nearly zero (less than 1%) for Shuffle Sort. The reason is that the Shuffle Sort algorithm mainly generates sequential reads and writes for each thread, which can be well handled by thread-local Direct SATC and Private SATC. Compared with Spark, GNU Sort and Shuffle Sort on TriCache is faster by 1.55× and 3.62×, respectively.

Out-of-core Performance. When the memory quota is less than 256GB for the 150GB workload, TriCache can provide tens of times speedups over swapping, up to 39.3× for GNU Sort at 128GB memory and 57.8× for Shuffle Sort at 64GB memory. Meanwhile, the performance of Shuffle Sort with TriCache is up to 20.2× better than Spark at 32GB memory.

For the 400GB dataset, both algorithms keep executing out-of-core. Shuffle Sort on TriCache is faster than swapping by

[5] https://github.com/apache/spark [tag v3.2.0]

up to 43.6× at 32GB memory and outperforms Spark by up to 13.7× with the same amount of memory. GNU Sort based on swapping is 41.2× slower than TriCache Shuffle Sort with 512GB memory and about 128× slower when the memory quota is less than 256GB because of its sub-optimized algorithm and the limited performance of the OS page cache.

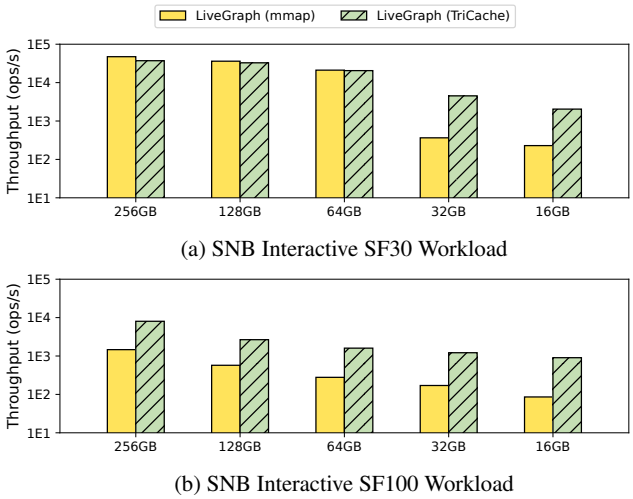
Compared with the in-memory processing of the 150GB dataset, Shuffle Sort with TriCache saves 90% memory with 32GB memory, while its processing time is only 49.3% longer than the processing time at 512GB. We also compared the distributed Spark and TriCache-based scale-up solutions. We use four servers with the same hardware configuration and connect them with 200Gb HDR Infiniband NIC. TriCache under 32GB memory outperforms in-memory distributed Spark by 7.20× using Shuffle Sort and 1.33× with GNU Sort on the 400GB workload. So TriCache with NVMe arrays can use less memory and provide nearly in-memory performance for TeraSort. In addition, TriCache can nearly utilize the peak bandwidth of our 8 NVMe SSDs, reaching 44GB/s for read-only operations and 31GB/s for mixed read/write operations.

In summary, developers can write in-memory programs (e.g., less than 20 lines of C++ code for Shuffle Sort), and TriCache then helps them to fully utilize the high-performance NVMe SSD array, especially when the algorithm is friendly to out-of-core processing.

4.4 Performance on Graph Database

For workloads in graph databases, we evaluate TriCache on LiveGraph[6] [57], an efficient transactional graph database based on OS memory-mapped files. LiveGraph treats memory-mapped files as in-memory data and relies on atomic memory accesses and cache consistency to support transactional queries. It can examine whether a user-transparent block cache is able to provide the same semantics as in-memory operations. We replace the memory-mapped files with TriCache and compare it with the original LiveGraph. We evaluate their performance on the LDBC SNB interactive benchmark, which simulates user activities in a social network and consists of 14 complex-read queries, 7 short-read queries, and 8 update queries. As the SNB driver occupies part of the memory, we limit LiveGraph to use up to 256GB memory and generate two workloads: SF30 and SF100 datasets. With LiveGraph, these datasets take about 100GB and 320GB memory, respectively. SNB clients request 1.28M operations for the SF30 workload, and 256K operations for the SF100 workload during the benchmark run.

Figure 12 shows the SNB throughputs of LiveGraph on TriCache and `mmap`. When the dataset can fit into a 256GB memory, the instrumentation and user-space cache of TriCache incur only 21% runtime overheads on the SNB benchmark. As the memory quota gradually decreases, the advantage of TriCache becomes increasingly prominent, e.g., TriCache outperforms `mmap` by 12.4× at 32GB memory as its scalable Shared Cache and the efficient I/O backend supply much higher throughputs. For SF100, LiveGraph keeps running in out-of-core states. TriCache improves the throughput by 5.48× compared with `mmap` at 256GB memory, and the speedup can grow up to 10.5× at 16GB memory.



(a) SNB Interactive SF30 Workload

(b) SNB Interactive SF100 Workload

Figure 12: Throughput of LiveGraph on TriCache and `mmap`

We then take a closer look at the latency metrics when running SF100 with 256GB of memory. TriCache cuts the average latency on complex queries by 11.5×, on short queries by 1.79×, and on update queries by 21.1× (geometric means). The P999 tail latency of TriCache keeps 10.9× lower than `mmap` on complex queries and 1.35× lower on short queries. Meanwhile, TriCache shortens the P999 latency of update queries to 34.6× shorter than the original LiveGraph because TriCache is additionally aware of thread locality while `mmap` is not. Although TriCache and `mmap` are both user-transparent, the Private SATC of TriCache can automatically hold recently updated data for writer threads in memory even when writers are waiting for group commits. On the contrary, `mmap` may evict these dirty pages under memory pressure. Our design helps LiveGraph to reduce tail latencies on update operations.

4.5 Micro-benchmarks

We conduct two custom multi-threaded micro-benchmarks which issue random memory-load instructions. The first generates random accesses in 8-bytes (named 8B Random workload), which can stress the systems in the case of completely random memory accesses. We control the random pattern to generate operations with different hit rates of block caches, and we also adjust the hit rate of Private SATC to examine its performance impact. The second randomly chooses 4KB pages and sequentially accesses each page in 8-byte words (named 4KB Random workload) to evaluate the performance when a page is accessed multiple times.

[6] https://github.com/thu-pacman/LiveGraph [commit eea5a40]

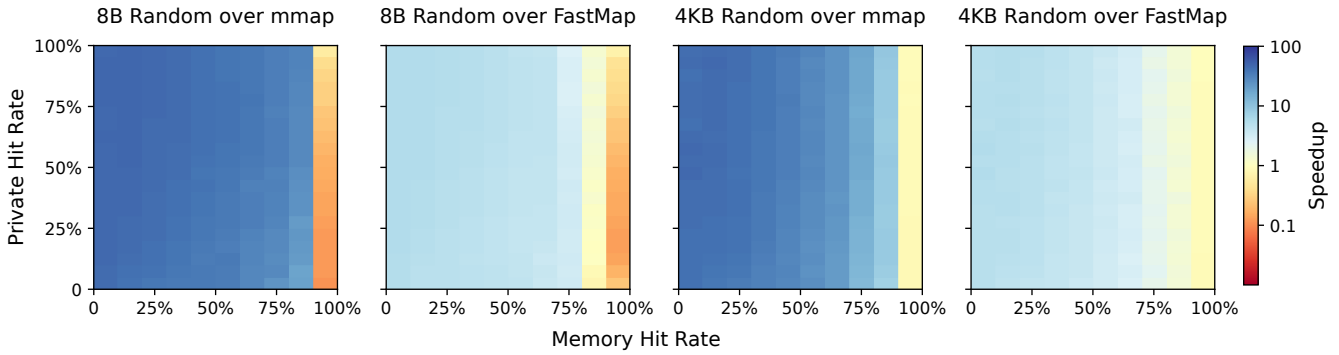


Figure 13: The speedup of TriCache compared with mmap and FastMap on 8B Random and 4KB Random workloads

We compare TriCache with Linux mmap and FastMap [29] (both given a hint of random accesses, MADV_RANDOM). FastMap optimizes the mmap path in the Linux kernel, including sharding locks (discussed in Section 3.2) and batching TLB invalidations. It mainly aim to mitigate the scalability limitation of Linux mmap. For FastMap, we downgrade the kernel version to 4.14 as FastMap relies on it, and configure a RAID-0 with mdadm as suggested by the authors of FastMap, and leverage FastMap to manage bare block devices.

Figure 13 shows TriCache's speedup compared with mmap and FastMap with 8B Random and 4KB Random workloads.

For 8B Random workloads, the performance of TriCache is about 11% of the in-memory (with mmap) performance in the worst case, when the memory hit rate is 100% and Private SATC hardly hits. Under the same 100% memory hit rate, when the hit rate of Private SATC grows up to 100%, TriCache can attain 57% of the in-memory (with mmap) performance, with a performance improvement of up to 6.07×.

Once the memory hit rate drops to 90%, TriCache can provide improvements of 18.6× to 31.5× over mmap whose performance is severely limited by lock contentions in the kernel. At the same time, TriCache outperforms FastMap by 1.22× on average. As the memory hit rate gradually decreases, the advantage of TriCache becomes increasingly significant. For instance, TriCache outperforms mmap by 33.6× and FastMap by 3.34× with an 80% hit rate. When the memory hit rate reaches 10%, TriCache performs 45.0× to 47.2× better than mmap, and 5.38× to 5.60× better than FastMap. In this case, TriCache provides 12.4 million random accesses per second and fully saturates our 8 NVMe SSDs. However, FastMap can only support 2.22 million accesses per second with all the hardware cores, where this is equivalent to about the I/O performance of only two NVMe SSDs. This indicates that FastMap cannot accommodate currently available high-performance NVMe SSD arrays because it still suffers from the heavy I/O stack of the kernel, page faults, and context switching overheads [27, 34, 55].

For 4KB Random workloads, Direct SATC can mainly absorb the in-memory overheads of TriCache. The performance of TriCache reaches 84% to 91% of the in-memory (with mmap) performance when the memory hit rate is 100%. With a 90% memory hit rate, TriCache can provide a speedup of 8.43× on average over mmap and 1.46× over FastMap. Under a 10% memory hit rate, TriCache can provide 12.3 million random accesses per second, which outperforms mmap by 43.1× and FastMap by 5.08× on average.

Table 1: Performance slowdown relative to TriCache

	Linux AIO	W/O Direct	W/O Private	Shared Only
PageRank	1.15×	2.75×	1.03×	40.1×
RocksDB	2.16×	1.27×	1.02×	22.0×
ShuffleSort	1.69×	1.87×	4.67×	10.1×
GNUSort	2.36×	2.51×	4.25×	57.9×
LiveGraph	1.21×	1.07×	1.01×	7.55×

4.6 Performance Breakdown

We select five cases under 64GB of memory for the breakdown experiments: PageRank, RocksDB, Shuffle and GNU Sort for the 400GB Terasort dataset, and LiveGraph for the SNB SF100 workload.

SPDK and Linux AIO. TriCache currently supports SPDK and Linux AIO as its storage backend. In the default configuration, TriCache uses SPDK to handle I/O operations. We compared the performance of these two backends. The first column in Table 1 shows the performance slowdown when using the AIO backend compared with the SPDK backend.

In terms of the (geometric) average, SPDK performs 1.64× better than Linux AIO, demonstrating that the user-space NVMe driver enables better IO performance. Nevertheless, SPDK has some drawbacks, such as high programming complexity, deployment difficulties, and not easy for supporting multiple applications. Luckily, TriCache hides SPDK programming details from users, allowing users to code in-memory programs and achieve efficient out-of-core performance. Moreover, the design of TriCache is not coupled with SPDK and can provide comparable performance with Linux AIO. If using or deploying SPDK is not feasible, AIO can serve as a reasonable alternative backend for use in TriCache.

Breakdown Analysis of SATC. The three columns on the right side of Table 1 break down the performance impact of SATC on TriCache by gradually removing SATC levels. *W/O Direct* disables Direct SATC, *W/O Private* disables Private SATC, and *Shared Only* uses only Shared Cache by removing both SATC levels.

According to the performance degradation listed in Table 1, SATC is an essential component contributing to the good performance of TriCache. The slowdown that occurs by disabling SATC (*Shared Only*) is 20.8× on average for the five cases. And even when the memory quotas are less than 1/5 of the working set (i.e., running out-of-core), SATC still yields a speedup of 40.1× for PageRank, 10.1× for Shuffle Sort, and 57.9× for GNU Sort.

Meanwhile, both Direct SATC and Private SATC are indispensable to TriCache. Without Direct SATC, the performance of PageRank is degraded by 2.75× because accessing each edge incurs a heavy overhead due to hash table lookups and evict policy maintenance. However, PageRank is not sensitive to Private SATC because the size of the dataset is more than 5× larger than the available memory, and the edges are visited only once for each iteration. For the shuffle phase of Shuffle Sort, the performance drops by 5.39× without Private SATC but remains almost the same (only 4.8% slower) without Direct SATC. The reason is that string copies constitute the bottleneck in the shuffle phase and are optimized by the compiler to `memcpy`, which is implemented by manually calling `pin`/`unpin` in the TriCache runtime. For GNU Sort, removing Direct SATC and Private SATC degrades the performance by 2.51× and 4.25×, respectively.

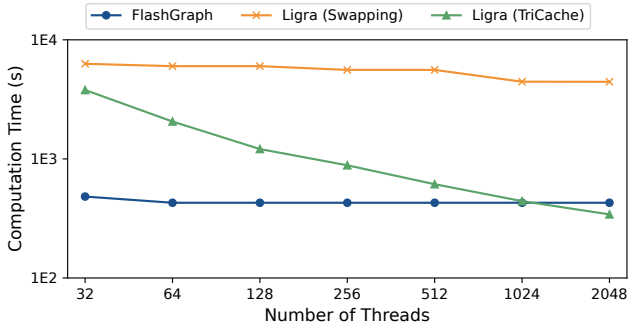
Multi-level Cache in TriCache. Next, we use PageRank, Shuffle Sort, and GNU Sort to further examine the design of the multi-level cache in TriCache. Table 2 lists the miss rates for each level of the cache, the average hit cycles (*HitC.*) for Direct SATC and Private SATC, and the average access cycles (*Acc.C.*) for Shared Cache.

According to the miss rates listed in Table 1, Direct SATC and Private SATC can handle most memory accesses. The miss rate of Direct SATC is less than 5% for all the three workloads, and the miss rate of Private SATC is less than 1% for Shuffle Sort and GNU Sort. The results show that SATC can cover most accesses to meet the above performance.

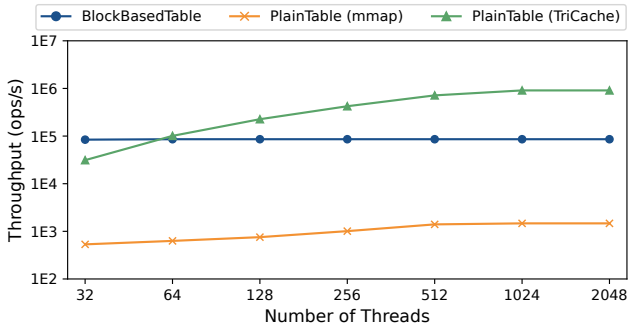
And the hit cycles of Direct SATC and Private SATC in Table 1 show that the software address translation of TriCache is quite efficient. The average costed cycles of Direct SATC hits in PageRank and Shuffle Sort are approximately 50 cycles; Direct SATC hits in GNU Sort and Private SATC hits in Shuffle Sort take about 150 cycles; Private SATC hits in GNU Sort use about 450 cycles. To give an idea of how much time they take, we list some hardware latencies: 50 cycles are close to a NUMA-local L3 cache hit or an L2 cache false sharing within a NUMA node; 150 cycles correspond to about the half of a NUMA-local memory access; 450 cycles are less than a cross-NUMA memory access or a cross-NUMA cache false sharing. Therefore, TriCache with SATC is efficient enough to provide a virtual memory interface and also to deliver memory-comparable performance.

Table 2: Miss rate and average cycles on each cache level

	Direct SATC		Private SATC		Shared Cache	
	MissRate	HitC.	MissRate	HitC.	MissRate	Acc.C.
PageRank	0.003	52.6	0.063	321	0.626	2.36M
ShuffleSort	0.001	63.0	0.001	162	0.969	1.68M
GNUSort	0.045	143	0.007	488	0.926	789K



(a) Computation time of PageRank (lower is better)



(b) Throughput of RocksDB

Figure 14: Performance of TriCache and baselines under different numbers of threads

Performance and Numbers of Threads. We also compare the performance of TriCache and baselines under different numbers of threads for PageRank and RocksDB with 64GB of memory. More precisely, "the performance under a given number of threads" means the maximum performance with less than or equal to this number of threads (only searched over powers of two). Since TriCache uses 16 server threads as the default configuration in the evaluation section, the number of threads starts with 32 threads (including server threads).

As shown in Figure 14, TriCache achieves good scalability for the workloads of both PageRank and RocksDB, which is one of the reasons for why TriCache performs well. For example, from 32 threads to 256 threads (the number of hard-

ware threads), Ligra with TriCache (in Figure 14a) achieves a 4.29× speedup, and RocksDB with TriCache (in Figure 14b) yields a 13.5× performance improvement.

Meanwhile, with a small number of threads, TriCache's performance is worse than that of manually optimized prefetch and asynchronous I/O because of TriCache's synchronous scheme for triggering I/O and its lack of program-specific optimizations (similar to `mmap`). In order to mitigate these limitations, over-subscription can help to utilize the queue depth of SSDs as much as possible. Through over-subscription, the performance of TriCache is improved by 2.58× for PageRank and 2.15× for RocksDB, thus enabling good performance for TriCache even without manual optimizations.

5 Related Work

There is a series of work that tries to improve the page caching performance with customized memory-mapped file I/O paths or swapping approaches [27–30, 39, 41]. Kmmap [28] provides several improvements to reduce the variation in performance owing to the aggressive write-back policy of Linux. FastMap [29] addresses scalability issues by separating clean and dirty pages and using per-core data structures to avoid centralized contentions, with the help of a custom Linux kernel. Still, our evaluation shows that FastMap cannot saturate current high-performance NVMe SSD arrays. Aquila [27] offers a library OS solution that eliminates the need for kernel modifications, relying on hardware support for virtualization, which makes it not easy to deploy on cloud environments. Umap [30] provides an `mmap`-like interface to user-space page fault handlers based on `userfaultfd` [2] in Linux, but is faster than `mmap` only with large page sizes. LightSwap [55] redesigns the swapping system to reduce context switching and page fault overheads, but it requires both kernel and program modifications. TriCache exposes a memory interface like these kernel-involved solutions, but runs completely in the user space to achieve maximal out-of-core performance.

Block caches (or buffer managers) are critical components in data-intensive applications for supporting out-of-core processing [6, 12, 15, 20, 52, 54]. Some attempts try to improve the performance of block caches. SAFS [53], the storage backend for FlashGraph [54], adopts a lightweight cache design based on NUMA-aware message passing. Users need to program with its asynchronous I/O interface to exploit maximal I/O performance on SSD arrays. LeanStore [20] proposes to use pointer swizzling so that pages residing in memory can be directly referenced without page lookups. However, it requires pages to form a tree-like structure, and thus is applicable to limited scenarios. TriCache shares similar goals but provides a memory interface that is user-transparent and more general.

Remote cache systems [14, 24] have been developed upon ideas of the disaggregated architecture [3, 11, 13, 18, 19, 25, 31, 34–36], which utilizes high bandwidth and low latency of modern networks. In this paper, we focus on scaling-up through NVMe SSD arrays. And we intend to consider support for disaggregated architectures in our future work.

Non-volatile memory (NVM) enables larger capacity compared with DRAM, and researches have been devoted to memory management instead of paging strategies to render memory access efficient on hybrid NVM and DRAM architectures [17, 32, 42, 56]. Nevertheless, block caches such as TriCache are still better suited for NVMe SSDs due to their higher latencies than NVM or DRAM.

6 Discussion

In TriCache, SATC does not need to be notified by its Shared Cache when a block is swapped out. In contrast, hardware TLB in processors, which also accelerates address translation as SATC, requires OS page cache to explicitly invalidate evicted pages through *TLB shootdown*, incurring considerable overhead [4, 5] due to inter-processor interrupts (IPIs). A comparison of the mechanisms of SATC and TLB shows that SATC utilizes reference counting to prevent evicting blocks currently being used by clients, while the OS is not directly aware of how many TLB entries are still referring to the pages to be evicted. It is possible to extend the design of SATC to TLB. Processors could mark the reference counts for page tables entries (PTEs), e.g., recording the number of TLB entries that currently hold a specific PTE. The OS can then adapt its page swapping and evict policies to avoid evicting pages currently present in TLBs, thus mitigating the performance issue brought by TLB shootdown.

7 Conclusion

In this paper, we explore a new user-space approach to achieving efficient out-of-core processing with in-memory programs, by providing a virtual memory interface on top of a block cache. We implement TriCache based on a novel multi-level design and applies it to various in-memory or `mmap`-based programs without manual code modification. TriCache achieves out-of-core performance that is orders of magnitude higher than that of the Linux OS page cache, and is often comparable to or even faster than specialized out-of-core solutions.

The open-source implementation of TriCache and instructions to reproduce the main experimental results are accessible from: `https://github.com/thu-pacman/TriCache`.

Acknowledgments

We sincerely thank Liuba Shrira (our shepherd) and all the anonymous OSDI and OSDI AE reviewers for their insightful comments and suggestions. This work was partially supported by National Key Research & Development Plan of China under grant 2017YFA0604500 and NSFC U20B2044. The corresponding author is Wenguang Chen.

References

[1] Sort Benchmark Home Page. http://sortbenchmark.org. [Online; accessed 31-May-2022].

[2] Userfaultfd — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html. [Online; accessed 31-May-2022].

[3] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATc} 18)*, pages 775–787, 2018.

[4] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 27–39, Santa Clara, CA, July 2017. USENIX Association.

[5] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down TLB shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, pages 1–14, New York, NY, USA, April 2020. Association for Computing Machinery.

[6] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with c++, 2016.

[7] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, May 2004. Association for Computing Machinery.

[8] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 587–596, New York, NY, USA, March 2011. Association for Computing Machinery.

[9] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, dec 2008.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[11] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. *ACM SIGCOMM Computer Communication Review*, 45(4):551–564, 2015.

[12] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[13] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 249–264, 2016.

[14] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 649–667, 2017.

[15] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a database system*. Now Publishers Inc, 2007.

[16] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[17] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706, 2015.

[18] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 1–15, New York, NY, USA, April 2016. Association for Computing Machinery.

[19] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 185–196, 2018.

[21] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 447–461, Huntsville, Ontario, Canada, October 2019. Association for Computing Machinery.

[22] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell+: Snapshot isolation without snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 425–441. USENIX Association, November 2020.

[23] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment*, 13(7):1091–1104, March 2020.

[24] Shuang Liang, Ranjit Noronha, and Dhabaleswar K Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *2005 IEEE International Conference on Cluster Computing*, pages 1–10. IEEE, 2005.

[25] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[26] Zhiyuan Lin, Minsuk Kahng, Kaeser Md Sabrin, Duen Horng Polo Chau, Ho Lee, and U Kang. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 159–164. IEEE, 2014.

[27] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped I/O on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 277–293. Association for Computing Machinery, New York, NY, USA, April 2021.

[28] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 490–502, 2018.

[29] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 813–827. USENIX Association, July 2020.

[30] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78. IEEE, 2019.

[31] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996.

[32] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.

[33] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 342–358, New York, NY, USA, October 2017. Association for Computing Machinery.

[34] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}: High-performance, application-integrated far memory. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 315–332, 2020.

[35] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[36] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[38] Johannes Singler and Benjamin Konsik. The gnu libstdc++ parallel mode: Software engineering considerations, 2008.

[39] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped i/o on fast storage device. *ACM Transactions on Storage (TOS)*, 12(4):1–27, 2016.

[40] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[41] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. Di-mmap—a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, 2015.

[42] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1541–1555, 2018.

[43] Wikipedia contributors. Memory-mapped file — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Memory-mapped%20file&oldid=1089594834, 2022. [Online; accessed 31-May-2022].

[44] Wikipedia contributors. Memory paging — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Memory%20paging&oldid=1068326108, 2022. [Online; accessed 31-May-2022].

[45] Wikipedia contributors. NVM Express — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=NVM%20Express&oldid=1090339430, 2022. [Online; accessed 31-May-2022].

[46] Wikipedia contributors. Page cache — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Page%20cache&oldid=1068818367, 2022. [Online; accessed 31-May-2022].

[47] Wikipedia contributors. Pci express — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=PCI_Express&oldid=1090153203, 2022. [Online; accessed 31-May-2022].

[48] Wikipedia contributors. U.2 — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=U.2&oldid=1066844795, 2022. [Online; accessed 31-May-2022].

[49] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-All replacement for a multilevel cache. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. USENIX Association.

[50] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. Spdk: A development kit to build high performance storage applications, 2017.

[51] Idan Yaniv and Dan Tsafrir. Hash, Don't Cache (the Page Table). *ACM SIGMETRICS Performance Evaluation Review*, 44(1):337–350, June 2016.

[52] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[53] Da Zheng, Randal Burns, and Alexander S. Szalay. Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[54] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. FlashGraph: Processing Billion-Node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.

[55] Kan Zhong, Wenlin Cui, Youyou Lu, Quanzhang Liu, Xiaodan Yan, Qizhao Yuan, Siwei Luo, and Keji Huang. Revisiting swapping in user-space with lightweight threading, 2021.

[56] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2195–2207, 2021.

[57] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment*, 13(7):1020–1034, March 2020.

Tiger: Disk-Adaptive Redundancy Without Placement Restrictions

Saurabh Kadekodi* *Google*
Francisco Maturana* *Carnegie Mellon University*
Sanjith Athlur *Carnegie Mellon University*
Arif Merchant *Google*
K. V. Rashmi *Carnegie Mellon University*
Gregory R. Ganger *Carnegie Mellon University*

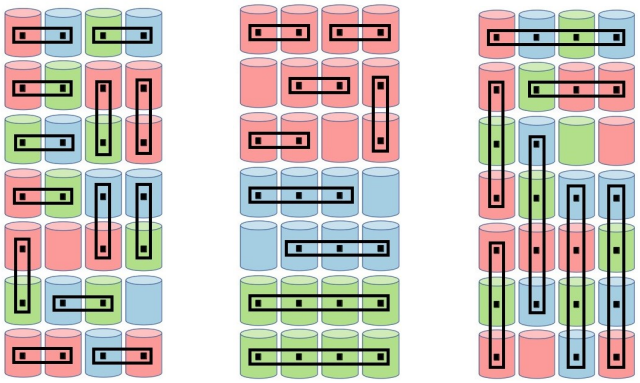
Abstract

Large-scale cluster storage systems use redundancy (via erasure coding) to ensure data durability. Disk-adaptive redundancy—dynamically tailoring the redundancy scheme to observed disk failure rates—promises significant space and cost savings. Existing disk-adaptive redundancy systems, however, pose undesirable constraints on data placement, partitioning disks into subclusters that have homogeneous failure rates and forcing each erasure-coded stripe to be entirely placed on the disks within one subcluster. This design increases risk, by reducing intra-stripe diversity and being more susceptible to unanticipated changes in a make/model's failure rate, and only works for very large storage clusters fully committed to disk-adaptive redundancy.

Tiger is a new disk-adaptive redundancy system that efficiently avoids adoption-blocking placement constraints, while also providing higher space-savings *and* lower risk relative to prior designs. To do so, Tiger introduces the *eclectic stripe*, in which redundancy is tailored to the potentially-diverse failure rates of whichever disks are selected for storing that particular stripe. With eclectic stripes, pre-existing placement policies can be used while still enjoying the space-savings and robustness benefits of disk-adaptive redundancy. This paper introduces eclectic striping and Tiger's design, including a new mean-time-to-data-loss (MTTDL) approximation technique and new approaches for ensuring safe per-stripe settings given that failure rates of different devices change over time. In addition to avoiding placement constraints, evaluation with logs from real-world clusters shows that Tiger provides better space-savings, less bursty IO for changing redundancy schemes, and better robustness (due to increased risk-diversity) than prior disk-adaptive redundancy designs.

1 Introduction

"A Tiger never changes its stripes", but can it be made to? In this context, the Tiger is a cluster storage system and its *stripes* are the erasure coded data that is placed across multiple disks in order to ensure data reliability. In today's cluster storage systems, most of the data reliability is via erasure coding [13, 21, 37, 40, 50, 58].



(a) Conventional cluster storage **(b)** Pacemaker (subcluster-based) **(c)** Tiger (this paper)

Figure 1: Stripe placements and configurations in different erasure coding systems: Disks of same color have similar annualized failure rates (AFRs), with red being least reliable (highest AFR), then blue, then green. Rectangles represent stripes with shorter stripes having higher redundancy. Conventional one-scheme-fits all designs (1a) impose no placement restrictions, but make no distinction of disk AFRs and therefore overprotect much of the data—all stripes use the widest redundancy scheme, shown as 2-wide for illustration. Pacemaker (1b) and Tiger (1c) tailor redundancy based on disk AFRs, resulting in different stripe widths in the illustration, and thereby reduce storage overhead. Pacemaker does this with rigid AFR-based subcluster boundaries, whereas Tiger requires no such boundaries.

Conventionally, a single cluster-wide redundancy scheme is selected for each data corpus (or for all data corpuses) [11, 14, 15, 21, 33]. This approach fails to account for the disk-reliability heterogeneity present in modern storage clusters, which consist of hundreds-of-thousands of hard disk drives (HDDs or just "disks") of multiple makes/models deployed at different times. This forces conventional storage clusters to use excessive redundancy (wasting capacity, and thus money and energy) to guarantee data safety, given that different disks have different failure rates. Absent other information, redundancy schemes are usually chosen to be safe for stripes fully stored on the least reliable disks (e.g., Fig. 1a). Recent research has showed that adapting redundancy scheme selection

*Equal contribution

to the observed failure rates of specific disks can reduce the space (=cost) overhead of redundancy by up to 20% [23].

Existing disk-adaptive redundancy designs [24, 25], however, face several significant adoption hurdles. At their core, these designs rigidly partition a storage cluster into subclusters of disks (called redundancy groups or Rgroups) that have similar failure rates, so they can use a subcluster-wide redundancy scheme tailored to meet the required data reliability target (e.g., Fig. 1b). Key adoption hurdles include: (1) Since each stripe must be entirely within a single Rgroup, this subcluster-based design can interfere with other data placement considerations, such as enhancing risk-diversity by spreading data across fault domains and different makes/models/batches of disks. Indeed, many of the Rgroups consist of a single make/model. (2) To provide reasonable degrees of performance and reconstruction speed scalability, subclusters must be sizable, making these designs only suitable for very large storage clusters. (3) When failure rates rise for a given make/model, as it ages, the redundancy scheme for an entire Rgroup (potentially 100s of PBs) may need to change to maintain target data reliability levels—all at once. The Pacemaker design [24] proposes to predict such changes and start them early, but they need to predict a month or more in advance to avoid reliability problems given the huge amount of data being transitioned, which is inherently a risky proposition. (4) The subcluster-based designs assume full adoption of disk-adaptive redundancy, not allowing for selective adoption for some data corpuses but not for others.

We present Tiger, a disk-adaptive redundancy system that eliminates the placement constraints posed by subcluster-based disk-adaptive redundancy designs while providing equal or greater benefits. Tiger's core new abstraction is the *eclectic stripe*, in which disks of different AFRs can be used to store a stripe that has redundancy tailored to the set of AFRs for those disks. In terms of placement flexibility, eclectic stripes are identical to stripes in conventional (non-disk-adaptive redundancy) designs. But, unlike conventional stripes, eclectic stripes do not conservatively assume the worst-case AFR for all disks. Instead, with eclectic stripes, the redundancy scheme is dynamically set for each stripe based on the AFRs of the chosen disks (e.g., Fig. 1c). Tiger's eclectic stripe approach avoids all the adoption hurdles discussed above, while simultaneously increasing the effectiveness (higher space-savings) and robustness (lower burstiness of urgent transition IO) of disk-adaptive redundancy.

Efficiently incorporating the proposed new abstraction of eclectic stripes is challenging due to multiple reasons. Tiger introduces several new design elements to overcome these challenges. First, calculating the exact reliability in terms of mean-time-to-data-loss (MTTDL) of a stripe can be prohibitively expensive, since accounting for different failure rates can lead to an exponential number of states in the traditional Markov chain reliability model. To address this, we provide a novel approximation technique that speeds up MTTDL calculation by 2-4 orders of magnitude while always preserving accuracy of over 95%, and on average over 99.5%. Second, while disks for a stripe can be chosen based on pre-existing placement policies, the chosen disks may not form an adequately-reliable stripe for a planned redundancy scheme, since the reliability is dependent on the chosen disks' AFRs. Tiger uses an AFR-aware stripe-width-reduction policy to quickly achieve sufficient reliability. Third, disk AFRs change over time [25], which can require changing the redundancy schemes of some eclectic stripes. Keeping track of AFRs for each stripe and triggering the redundancy schemes can significantly increase the overhead for metadata and background operations. Tiger introduces an *eclectic volume* abstraction to reduce metadata overhead and make identification of required changes efficient. It also introduces policies to reduce transition IO: the IO involved with enacting changes to stripe redundancy schemes.

Evaluating the feasibility and efficacy of eclectic stripes requires analysis of long-term effects on huge storage clusters. We evaluate Tiger using the same logs as used to evaluate Pacemaker [24], enabling an apples-to-apples comparison. These logs contain all disk-deployment, failure, and decommissioning events from four production storage clusters: three 160K–450K-disk Google clusters and a ≈110K-disk cluster used for the Backblaze Internet backup service [3]. Simulation driven by production logs allows us to analyze reliability, space usage, and redundancy maintenance traffic for multiple clusters each with over 100K disks and over multiple years, which would be infeasible otherwise as part of a research setup. For all four clusters, Tiger provides equal or better space-savings than Pacemaker, while requiring at most 0.5% of daily IO bandwidth for transition IO. More importantly, the transition IO is both less bursty, in terms of when it is needed, and less urgent, in terms of how unsafe an unsafe stripe might be if the scheme transition were delayed. For instance, in response to a tiny rise in AFR ($< 0.25\%$) for disks of a given make/model, Pacemaker would need 196% of the total IO bandwidth from each of those disks in order to make the data safe—to avoid stealing more than 5% of IO bandwidth for transition IO, Pacemaker would have to know to start 40 days in advance—but Tiger would need $<$1.6% even for a 1% AFR increase because of the diversity of its eclectic stripes. And, most importantly, Tiger exhibits significantly better risk-diversity, stemming from removing placement constraints and allowing differently-reliable disks (and hence disks of different makes/models) to belong to the same stripe. For example, even with random selection of disks for each stripe, most of Tiger's eclectic stripes span most of a cluster's make/models; Pacemaker's strict Rgroup boundaries disallow use of more than one make/model for most stripes.

Contributions. In this paper, we make four main contributions. First, we introduce eclectic stripes as a tool for realizing disk-adaptive redundancy without the placement restrictions posed by prior designs. Second, we present a reliability model

and its approximation to efficiently calculate the MTTDL of eclectic stripes. A surprising outcome is that a homogeneous stripe with the same scheme and average disk AFR as an eclectic stripe is less reliable! Third, we present the design and architecture of Tiger, the first disk-adaptive redundancy system for supporting and efficiently managing eclectic stripes. Fourth, we evaluate Tiger and compare it to the state-of-the-art, using logs from four large real-world storage clusters, demonstrating its effectiveness in realizing disk-adaptive redundancy without prior designs' adoption challenges and with greater space-savings and lower risk.

2 Background and Motivation

We first provide a primer on data redundancy done using erasure coding followed by the gist and importance of disk-adaptive redundancy. We then describe the problems with existing disk-adaptive redundancy systems, which is the motivation for this paper.

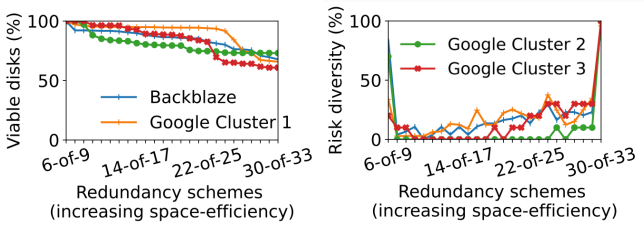
Erasure Coding for data durability. Modern storage clusters often comprise of hundreds-of-thousands of disks of multiple make/models deployed over time. The sheer scale of the storage clusters makes disk failures a common occurrence [15], which necessitates some form of redundancy to ensure data durability and availability. While replication is popular for availability of hot data, erasure coding (a more space-efficient alternative to replication) is more common for the durability of colder data, which forms the majority of the stored data. In erasure coding (EC), data is split into k chunks, and $n-k$ *parity* chunks are subsequently generated to form a *stripe* with n chunks. Each chunk is stored on a separate disk. This k-of-n EC scheme (also called "redundancy scheme") can withstand up to $n-k$ failures with a storage overhead of $\frac{n}{k}$. Any k chunks of an n-chunk stripe are sufficient to construct the original data.

Reliability Metrics: MTTDL and AFR. The reliability of a stripe is determined by its *mean-time-to-data-loss* (MTTDL). A stripe's MTTDL is calculated using a continuous-time Markov chain shown in the left side of Fig. 3. Each state represents the number of simultaneously lost chunks in a stripe. The MTTDL is the mean time to reach state DL (where $n-k+1$ chunks are simultaneously lost) from state 0; this is when data is irrecoverably lost. This model assumes a *homogeneous stripe*, where all disks fail with the same rate λ. Downward transitions denote failures, which happen with a rate of λ times the number of available chunks, while upward transitions denote repairs, which happen with a fixed rate μ. Failure rates are commonly expressed as an *annualized failure rate* (AFR), which is defined as the expected fraction of failed disks in a year, assuming that failed disks are replaced and the disk population remains fixed.

Disk-adaptive redundancy. Storage clusters have conventionally been using a one-scheme-fits-all redundancy scheme by assuming that all disks fail similarly. Prior work has shown that disk AFRs are highly correlated with their vintage [26, 35]. With modern clusters having a mix of disk makes/models/batches, there can be over an order of magnitude difference between AFRs of different groups of disks [25]. Additionally, over their lifetime, disk AFRs follow a "bathtub curve" with multiple failure regimes: infancy (high AFR) followed by useful life with potentially multiple phases (piecewise linear phases with low AFR that increases gradually) and finally wearout (high AFR) [24].

Disk-adaptive redundancy capitalizes on differences in disk AFRs and dynamically tailors data redundancy to observed disk failure rates [23]. Disk-adaptive redundancy systems take into account various constraints including the reconstruction costs when making the decision of a target stripe width to adapt to. Specifically, wide schemes are used only when a stripe's average AFR is low enough to keep the reconstruction cost contained below a configured limit. More generally, wide stripes provide cost savings in terms of smaller storage overhead at the cost of higher reconstruction costs and higher degraded mode reads. We know from conversing with architects of large-scale storage clusters that the cost of the excess byte footprint matters more than the cost of excess IO required in the context of redundancy, given existing workloads. This is especially so since, in general, large-scale capacity-tier storage cluster workloads tend to be cold (have low IO/s per byte). Additionally, cold data experiences fewer reads, and therefore has very few costly degraded mode reads. Backblaze is an example where, for archival data that has low IO access rates, administrators have publicly confirmed use of wide redundancy schemes such as 17-of-20 [4]. By using more space-efficient redundancy schemes during low AFR regimes, disk-adaptive redundancy can provide substantial space-savings ($> 20\%$) in clusters with over 100K disks.

Prior disk-adaptive redundancy systems. Two disk-adaptive redundancy systems have been proposed in the literature: HeART [25] and Pacemaker [24]. In HeART, the authors propose a tool to statistically learn the AFRs of different disk groups and identify change-points for safe redundancy transitions. By transitioning to an encoding scheme with minimum storage overhead that still meets the target MTTDL, HeART was able to obtain $\approx 20\%$ space-savings when tailoring erasure codes, and $\approx 33\%$ space-savings when tailoring replication. Although lucrative, HeART overlooked an important practical hurdle in performing disk-adaptive redundancy: *transition overload*, i.e. the IO overhead of performing redundancy transitions. Crippling transition overload when thousands of disks require simultaneous redundancy transitions forms the basis for Pacemaker [24]. The gist of Pacemaker is to convert urgent redundancy transitions into schedulable ones by making conservative predictions of the rise in AFR and proactively issuing redundancy transitions. This allows the transition overload to be spread out over time, such that it can be completed within tolerable IO limits without compromising data safety.



(a) Placement constraints **(b)** Risk-diversity

Figure 2: 2a shows Pacemaker's placement constraints by highlighting the fraction of the disk fleet that is viable for different schemes exercised on four production clusters. Fig. 2b shows the risk-diversity obtained by the same clusters on particular dates in their lifetime. A risk-diversity of 100% implies at least one chunk stored on every possible make/model, whereas a 0% risk-diversity implies that the particular scheme was not feasible in the cluster. Pacemaker performs poorly in both placement constraints and risk-diversity.

2.1 Existing designs are impractical

Despite remarkable space-savings and low IO costs, existing disk-adaptive redundancy systems remain impractical in real-world settings.

Placement restrictions. The primary hurdle stems from the placement restrictions posed by reliance on redundancy groups (*Rgroups*). An Rgroup is a set of disks with similar AFRs, such that they can use the same redundancy scheme. Prior systems redundancy management techniques rigidly partition the cluster's disks into Rgroups, and every stripe must be stored entirely within a single Rgroup. Fig. 2a shows the percentage of disks that are rendered infeasible for various redundancy schemes Pacemaker can employ on a particular day in four large storage clusters. More than 30% of the disks are deemed infeasible for space-efficient schemes beyond 22-of-25, because their AFRs are not low enough for those disks to participate in an Rgroup for which schemes beyond 22-of-25 can meet the target MTTDL. Furthermore, in order to maintain proper redundancy, stripes are typically constrained to span across different racks, servers, power lines, etc. Adding another placement constraint may be close to impossible.

Lower risk-diversity. Due to high correlation of AFRs and makes/models/batches [26,35], and in order to enable efficient transitioning mechanisms, many Rgroups contain disks from just one make/model. This is undesirable from a risk-diversity perspective. Fig. 2b shows the fraction of makes/models that are covered for the same stripe configurations in the same four clusters described above. Higher risk-diversity is valuable for mitigating consequences of bulk failure situations (e.g., from rapid degradation due to manufacturing defects), especially in a disk-adaptive redundancy system where redundancy is tuned rather than regularly excessive.

Reliance on AFR prediction. With lower risk-diversity, Pacemaker's Rgroups are already susceptible to data loss due to bulk failures in a single make/model (uncommon, but not impossible). Furthermore, Pacemaker's IO cost reduction is highly dependent on being able to accurately predict an AFR rise well in advance. Currently AFR is calculated only on the basis of age. Prior work has highlighted that it is dependent on various factors such as vintage, temperature, vibration, etc. [7, 26, 27, 35]. This makes an already difficult task of accurate AFR prediction even harder.

All-or-nothing. Current disk-adaptive redundancy designs depend on forming Rgroups, and work efficiently if entire Rgroups perform redundancy transitions together (for step-deployed disks). This implies that the entire cluster must commit to performing disk-adaptive redundancy for all of their data stored on all disks. Such a restriction makes disk-adaptive redundancy unusable without a major overhaul of the architecture of the existing storage cluster.

The key takeaway is that additional data placement restrictions create adoption-blocking limitations and risks. In order have have both placement flexibility and disk-adaptivity, we need a new approach that includes the ability to reason about and tune the reliability of stripes that span disks with different AFRs. We achieve this via ***eclectic stripes***.

3 *Eclectic Stripes* and their challenges

Eclectic stripes are central to Tiger's approach of providing disk-adaptive redundancy without placement restrictions. An eclectic stripe is an EC stripe placed on a collection of disks that can have different failure rates. The reliability model of conventional EC stripes forces them to be allocated on disks having (or worse, assumed to be having) the same failure rate. In terms of composition an eclectic stripe is no different than what a conventional EC stripe would be. Specifically, the same disks that make up a conventional stripe can also make up an eclectic stripe, just that eclectic stripes are cognizant of the AFR differences of the underlying disks and can accurately reason about the resulting reliability. A disk-adaptive redundancy system that supports eclectic stripes has to overcome several challenges.

1. Ensure efficient creation of sufficiently reliable eclectic stripes. Taking AFR differences of all disks in a stripe into account makes exact MTTDL calculation of eclectic stripes prohibitively expensive (see §4.1.1). Since stripe creation is a critical-path operation, it is imperative that a disk-adaptive redundancy system supporting eclectic stripes reasons about its reliability in an efficient and accurate manner.

2. Ensure efficient management of eclectic stripes. All underlying disks of an eclectic stripe will not experience an AFR rise or fall together. A system supporting eclectic stripes must efficiently identify which stripes need to change their redundancy in response to changing AFRs.

3. Support unchanged placement policies. While tweaking the placement policies might provide additional optimizations, a system that supports eclectic stripes must support existing placement policies without any change.

4. Retain key benefits of disk-adaptive redundancy. Dynamic redundancy adaptation at a low transition IO cost; continuously providing adequate reliability; providing space-savings by using more space-efficient redundancy schemes in low-AFR regimes are the key benefits of disk-adaptive redundancy. Any proposed disk-adaptive redundancy system should strive to maintain these benefits.

5. Ensure an adoption-friendly design. Apart from placement restrictions, existing disk-adaptive redundancy system designs require that the entire cluster commits entirely to perform disk-adaptive redundancy, or it cannot gain any of its benefits. Moreover, only the very large-scale storage clusters can use existing disk-adaptive redundancy designs, whereas the small and medium sized clusters are outside their scope. High emphasis on usability and showcasing a way for easy adoption of disk-adaptive redundancy in existing storage clusters of all shapes and sizes is an important design challenge.

4 Mechanisms to enable eclectic stripes

In this section, we address the two main challenges of eclectic stripes: their reliability and their management.

4.1 Interpreting reliability of eclectic stripes

We first shed light on key takeaways from our study of the reliability of eclectic stripes and then provide the detailed theory and the associated analysis.

Calculating MTTDL of eclectic stripes is efficient and accurate. The exact calculation of the MTTDL of an eclectic stripe is computationally expensive. We provide a novel approximation that provides the MTTDL with over 99.5% accuracy (on average), and always provides over 95% accuracy in our tests. In practice, a difference of 5% in MTTDL typically translates into a difference of around 0.1% AFR for a homogeneous stripe, which is negligible. The exact MTTDL calculation and the approximation are detailed in §4.1.1, 4.1.2.

Eclectic stripes are more reliable than homogeneous stripes. When comparing the MTTDL of an eclectic stripe with a homogeneous stripe having the same EC scheme and same avg. AFR, the MTTDL of the eclectic stripe is always higher than the MTTDL of the corresponding homogeneous stripe for typical system parameters (§4.2, Fig. 4).

Eclectic stripes are robust to AFR changes of individual disks. The MTTDL of the eclectic stripes does not react abruptly to the increase in AFR of a few disks. Compared to the conventional approach of treating stripes as homogeneous with AFR equal to the maximum AFR in the stripe, MTTDL of eclectic stripes react very gradually to AFR changes.

Eclectic stripes are more robust to AFR misestimations. Due to the nature of empirical data, any system that measures AFR has to estimate it. Since the AFRs of different disk make/models are estimated independently, it is unlikely that there will be simultaneous underestimation of the AFR of every disk in an eclectic stripe, and hence the impact of estimation errors is smaller (Fig. 5) and may even cancel each other out. Furthermore, disk-adaptive redundancy systems are made even more robust against misprediction by the use of confidence intervals. Thus, eclectic stripes are more robust to AFR misestimations compared to homogeneous stripes.

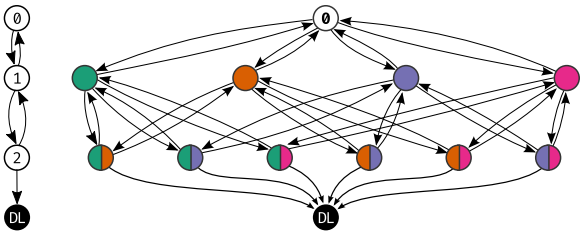


Figure 3: Left: Classic Markov chain model for the MTTDL of a 2-of-4 homogeneous stripe. Right: Markov chain model for the MTTDL of a 2-of-4 eclectic stripe.

4.1.1 Exact MTTDL calculation is costly

Using a Markov chain model to calculate the MTTDL of storage systems is a classic approach [16]. A generalization of this approach helps us take into account disks with different failure rates. Consider an EC stripe of a k-of-n scheme, placed over n disks with failure rates $\lambda_i (i \in [n])$ and a disk repair rate of μ. The state of the system is given by an n-length vector $\mathbf{s} = (s_0, \ldots, s_n)$ with $s_i = 1$ if disk i has failed, and $s_i = 0$ otherwise ($i \in [n]$). The state space is given by states $(s_i)_{i=1}^{n}$ such that the total number of failure $\sum_{i=0}^{n} s_i$ is at most the number of parities $n-k$, and a data loss state labeled DL. Therefore, the total number of states is $1 + \sum_{i=0}^{n-k} \binom{n}{i}$. The rate of transition from state $\mathbf{s}$ to $\mathbf{s}'$ is defined as:

- λ_i if $s_i = 0, s'_i = 1$, and $s_j = s'_j$ for $i \neq j$ (i^{th} disk fails),
- μ if $s_i = 1, s'_i = 0$, and $s_j = s'_j$ for $i \neq j$ (i^{th} disk repaired),
- $\sum_{i=1}^{n} (1 - s_i)\lambda_i$ if $\sum_{i=1}^{n} s_i = n - k$ and $\mathbf{s}' = DL$ (any disk fails when $n-k$ disks have failed and are not repaired).

The MTTDL is defined as the mean time to state DL from the initial state $\mathbf{0} = (0, \ldots, 0)$.

Given the values of $n, k, (\lambda_i)_{i=1}^{n}$, and μ, one can compute the MTTDL by using the standard approach of solving a system of equations. However, this approach is not tractable, due to the exponential explosion on the number of states with respect to $n-k$ (see Fig. 3 to compare conventional Markov chain with that of an eclectic stripe). For example, the Markov chain of a 10-of-14 eclectic stripe has 1472 states, compared to 6 states in the case of a 10-of-14 homogeneous stripe. Reasoning about this model can be hard too, since it is not directly clear how disk AFRs affect MTTDL. Furthermore, this approach tends to be numerically unstable, which makes obtaining precise MTTDLs hard. We find that computing a single MTTDL using this approach with realistic parameters can take up to several seconds using the Mathematica 12

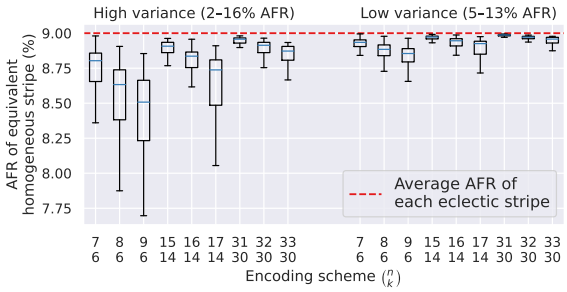


Figure 4: Reliability of eclectic stripes compared to homogeneous stripes. For each scheme, we sample 1000 eclectic stripes and for each stripe we compute its MTTDL ρ and then compute the AFR λ of a homogeneous stripe with the same scheme and MTTDL equal to ρ. The boxes show the distribution of λ over the 1000 stripes. The AFR of the first $n-1$ disks in a eclectic stripe are sampled uniformly at random from the range 2–16% (high variance) or 5–13% (low variance), and the AFR of the last disk in a stripe is chosen to ensure that the average AFR of the disks *in each stripe* is fixed at 9%. E.g. the median 6-of-9 eclectic stripe from the high-variance group is as reliable as a 6-of-9 homogeneous stripe with AFR 8.5%, despite having an average AFR of 9%.

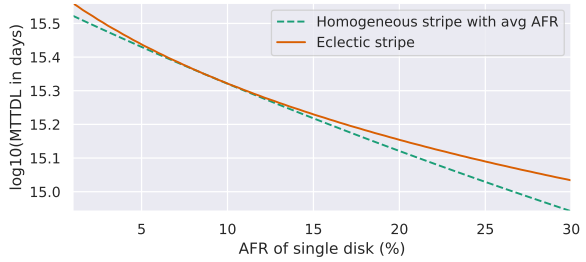


Figure 5: Reliability of a 6-of-9 eclectic stripe when the AFR of a single disk varies. The eclectic stripe is composed of 8 devices with AFR 9%, and one device whose AFR varies from 1% to 30% (x axis). The dashed line denotes the MTTDL of a 6-of-9 homogeneous stripe with the same average AFR as the eclectic stripe. The solid line denotes the MTTDL of the eclectic stripe. Reliability of the eclectic stripe is always above the corresponding homogeneous stripe.

software [52] on a desktop PC. This is too slow in practice, because not only do we need to compute the MTTDL when creating new stripes, but we also need to periodically compute the MTTDL of every stripe in the system (typically billions) as device AFRs change. The next section describes an efficient approximation that makes the MTTDL calculation of eclectic stripes computationally tractable and highly accurate.

4.1.2 Efficient and accurate MTTDL approximation

In order to compute and better understand the MTTDL of eclectic stripes, we propose an approximation formula, building on the approach presented in [2] for homogeneous stripes. This approximation is extremely good when $\mu \gg \max_i \lambda_i$, which is true for modern cluster storage systems.

The main idea behind this approximation is to note that (in the steady state) disk i will be available a fraction $A_i = \mu/(\mu+\lambda_i)$ of the time, and that the system will reach the *DL* state when exactly $k-1$ of the disks are available. Therefore, the MTTDL can be approximated with the following formula (see appendix A for the full derivation):

$$\text{MTTDL} \approx \left(\mu(n-k+1)\,\text{PBin}(k-1;n,(A_i)_{i=1}^n)\right)^{-1}, \quad (1)$$

where $\text{PBin}(k;n,(p_i)_{i=1}^n)$ is the probability of obtaining exactly k heads when flipping n biased coins with probability of heads p_i for coin i. PBin is known as the Poisson-binomial distribution, and it can be efficiently evaluated [12, 19].

We tested this approximation against the Markov chain approach over all values of $6 \le k \le 30$, $1 \le n-k \le 3$, and AFRs of 1–16%. The relative difference between the two output MTTDLs never exceeded 5% and was less than 0.5% on average[*]. As a benefit, the approximation is 2–4 orders of magnitude faster to evaluate (in the order of milliseconds), more numerically stable, significantly simpler to implement, and gives direct insight into how the parameters affect MTTDL.

4.2 Understanding MTTDL of eclectic stripes

The main difference between the reliability of an eclectic stripe and a homogeneous stripe is given by the Poisson-binomial factor in Eq. 1, which becomes Binomial when all probabilities are equal. Notice that the difference between A_i in Eq. 1 will be small because $\mu \gg \max_i \lambda_i$, and therefore the corresponding Poisson-Binomial distribution will not deviate too much from a Binomial distribution with trial success probability $A = \sum_{i=1}^n A_i/n$ [6]. Furthermore, we have:

$$\sum_{i=1}^n \frac{A_i}{n} = \frac{1}{n}\sum_{i=1}^n \frac{1}{1+\lambda_i/\mu} \approx \frac{1}{n}\sum_{i=1}^n \left(1-\frac{\lambda_i}{\mu}\right) = 1 - \frac{\sum_{i=1}^n \lambda_i/n}{\mu},$$

where we use the approximation $1/(1+x) \approx 1-x$ for small x. This means that the reliability of an eclectic stripe will tend to be close to the reliability of a homogeneous stripe with AFR equal to the average AFR of the eclectic stripe.

To measure how close the MTTDL of an eclectic stripe will be to that of a homogeneous stripe with the same scheme and average AFR, we conduct two numerical experiments. Fig. 4 compares eclectic stripes against homogeneous stripes that have the same MTTDL, across different schemes and AFR ranges. In this experiment, instead of directly showing an MTTDL ρ (which is hard to interpret) in the y-axis, we show the AFR λ of a homogeneous stripe that has MTTDL equal to ρ (under the relevant scheme). The results show that eclectic stripes are *more* reliable than homogeneous stripes with the same scheme and average AFR. In other words, for a homogeneous stripe composed of disks with AFR λ to match

[*]The median relative difference between the exact and approximated eclectic stripe MTTDL was 0.1%, the 90^{th} percentile error was 0.5%, and the 95^{th} percentile error was 0.7%.

the reliability of an eclectic stripe with AFRs $(\lambda_i)_{i=1}^n$, the disks in the homogeneous stripe have to be more reliable on average, i.e., $\lambda < \sum_{i=1}^n \lambda_i/n$. The difference, however, becomes small when the ratio n/k is small, or the range of AFRs is small. Fig. 5 shows the reliability of an eclectic stripe when the AFR of a single disk in the eclectic stripe varies in the range 1–30%. This experiment shows that eclectic stripes provide a dampening effect against AFR rises of a small number of devices in two ways: (1) a small number of devices have a smaller impact on the average AFR of the stripe (slope of the dashed line), and (2) the convex shape of the curve shows that the eclectic stripe is even more reliable than a homogeneous stripe with the same scheme and average AFR.

Checking if a stripe is safe: Typically, a minimum level of reliability is set in the cluster by setting a *MTTDL threshold* that all stripes must satisfy in order to be deemed safe. Given the results presented in this section, we now describe a simple method to determine whether a stripe is safe. We define the *critical AFR* of a k-of-n scheme and MTTDL threshold θ as the highest AFR that disks in a homogeneous k-of-n stripe can attain while still having an MTTDL of at least θ. The critical AFRs for the different schemes that are used in a system can be precomputed and stored. Then, a simple andx efficient way of checking whether an eclectic stripe under some scheme is safe is to check whether the average AFR in the stripe is less than the critical AFR for that scheme. Since an eclectic stripe is at least as reliable as a homogeneous stripe with the same scheme and average AFR, if the stripe passes this check, then we can be certain that the stripe is safe. If the stripe does not pass the check, then it *may* be unsafe, which can determined by computing its MTTDL. This test can help greatly reduce the amount of work needed in checking whether stripes are still safe, and it also provides a simple way of understanding the reliability of eclectic stripes.

4.3 Eclectic Volumes

Disk AFR changes may trigger redundancy transitions. Prior designs performed disk-adaptive redundancy at the disk level. Thus, if a disk's AFR changed, either all or none of the stripes on that disk required a redundancy transition. With eclectic stripes, each disk may store chunks of stripes with different reliabilities. An AFR change might only require redundancy transitions for a subset of those stripes. With millions of eclectic stripe chunks being stored on each disk, a linear search through all of them for each AFR change is impractical.

An eclectic volume is a collection of eclectic stripes that use the same EC scheme and are stored on the same set of disks. A disk can contain multiple volume fragments identified by their globally unique volume ID. Each disk maintains a map of stripe ID to eclectic volume ID. Since each eclectic volume spans the exact same disks, whenever a disk's AFR changes, Tiger only needs to check whether the EC scheme used for each of the disk's constituent volumes still meets the required

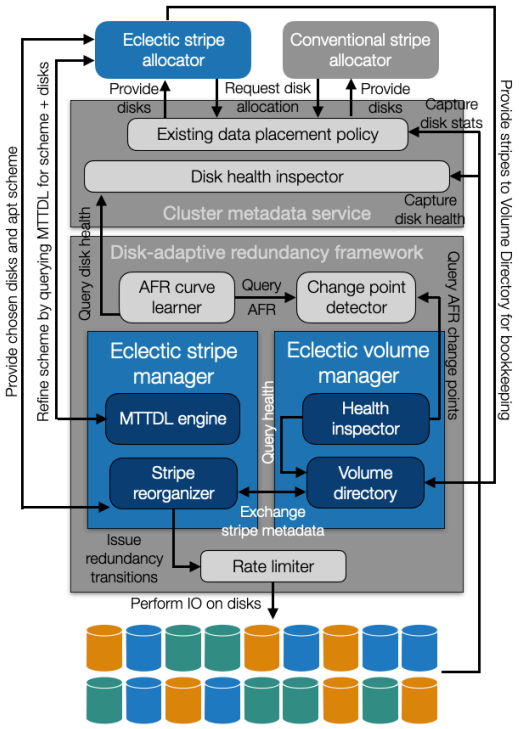


Figure 6: Architecture of Tiger. The blue boxes correspond to Tiger's components. The gray boxes correspond to existing components in cluster storage system architecture and components present in existing disk-adaptive redundancy systems.

MTTDL target. There is no need to check the reliability of each of the individual eclectic stripes within a volume since they are all identically reliable. The details of how Tiger manages eclectic volumes is described in §5.3.

Eclectic volumes prove to be efficient only if they represent a large number of eclectic stripes. Therefore, in Tiger the default size of an eclectic volume is set to 1 TeraByte (TB). This way, even though Tiger performs reliability monitoring at the volume granularity it ensures that each eclectic stripe is always sufficiently reliable.

5 Design and working of Tiger

Tiger is a practical disk-adaptive redundancy system designed to overcome the challenges described in §3. Fig. 6 shows the architectural components of Tiger (colored boxes) and how they interact with existing cluster storage system components and common disk-adaptive redundancy components.

5.1 Data flow in Tiger

We overview Tiger by explaining the lifecycle of eclectic stripes. An eclectic stripe is created via the *Eclectic Stripe Allocator* (ESAllocator), which identifies a set of disks and the corresponding scheme on which this data is to be stored. The ESAllocator uses the existing and unmod-

ified data placement policy to obtain a set of disks. That placement policy uses whatever knowledge designers choose (e.g., available freespace, load balance, and fault domain constraints) in selecting the set of disks. The ESAllocator then queries the *Eclectic Stripe Manager's MTTDL Engine* (ESMTTDLEngine) with the AFRs of the chosen disks, and a stripe configuration, to verify that the planned stripe's MTTDL meets the required target MTTDL. If it does not, the ESAllocator boosts the MTTDL by changing the stripe configuration until an appropriately safe redundancy scheme is found. §5.2 details this process.

Once created, the ESAllocator passes the stripe to the Eclectic Volume Manager (EVManager, see §5.3) to either add the stripe to an existing volume, or create a new volume which will contain the new stripe. The Eclectic Volume Health Inspector (EVHInspector) continuously monitors the reliability of the eclectic volume by querying the change point detector, which identifies significant AFR changes in the data from the AFR curve learner. The AFR curve learner, change point detector and the rate limiter can be reused without change from any existing disk-adaptive redundancy system[*]. In reaction to a significant AFR change (rise or fall), the EVHInspector alerts the EVManager, which fetches the eclectic stripe metadata from the EVDirectory and provides both the AFR change and the metadata to the Eclectic Stripe Reorganizer (ESReorganizer; see §5.2). The ESReorganizer includes techniques to efficiently perform redundancy transitions. If eclectic stripes must change, the ESReorganizer consults the ESAllocator in forming them. Non-urgent redundancy transitions (when the target MTTDL is not at risk of being violated) are throttled by the rate limiter in order to not overwhelm the storage cluster.

Tiger's stripe-by-stripe disk-adaptive redundancy approach enables incremental adoption by allowing data to be stored either as an eclectic stripe or a homogeneous stripe. This is in contrast to subcluster-based designs that are all-or-nothing.

5.2 The Eclectic Stripe Manager

The Eclectic Stripe Manager (ESManager) handles construction, maintenance and reorganization of eclectic stripes.

Constructing eclectic stripes In the absence of an existing eclectic volume that has space (described later in §4.3), the ESAllocator asks the existing data placement policy for disks to store each new eclectic stripe. Since that placement policy is unaware of disk-adaptive redundancy, it may return a set of disks whose AFRs produce an MTTDL that either fails to meet or far exceeds the target MTTDL. Algorithm 1 describes the process to build a space-efficient, yet adequately reliable eclectic stripe.

To give itself flexibility, ESAllocator asks the placement policy to provide a set of disks for the maximum-width-allowed stripe (e.g., 33 for 30-of-33). The ESAllocator then queries the ESMTTDLEngine with the provided disks and its planned scheme to get the MTTDL value. If the MTTDL does not meet the target MTTDL, ESAllocator discards a disk from the set and increases the redundancy of the corresponding scheme (e.g., 29-of-32 instead of 30-of-33) to boost the stripe's MTTDL, repeating this process until sufficient MTTDL is achieved. This process is guaranteed to terminate, since the least space-efficient scheme in a storage cluster must meet the target MTTDL. Moreover, by iterating from the most space-efficient scheme allowed, the algorithm terminates at the most space-efficient scheme for the provided disks.

Algorithm 1

$\theta_{\text{MTTDL}} \leftarrow$ target MTTDL
$n_{\max} \leftarrow max\{n \mid (n,k) \in \text{schemes}\}$
$(d_1,\ldots,d_{n_{\max}}) \leftarrow n_{\max}$ randomly sampled devices
for $(n,k) \in$ schemes in order of increasing n/k **do**
  if $\text{MTTDL}(n,k,(d_1,\ldots,d_n)) \geq \theta_{\text{MTTDL}}$ **then return** (n,k)

Ensuring reliability amid disk failures. The reliability of each eclectic stripe is a function of the AFRs on the disks on which it is stored. So, when a disk fails, the reconstructed data cannot simply be placed on a randomly chosen disk, since its AFR might be high enough to cause the eclectic stripe's MTTDL to exceed the target. Recall, from §4.2, that the critical AFR of an EC scheme is the highest AFR that a homogeneous stripe of that scheme can reliably support, and a simple way to test that an eclectic stripe is safe is to check that its average AFR is below the critical AFR for its EC scheme. Therefore, we can ensure that reliability will be preserved if we choose a disk that keeps the average AFR of the affected stripes under their respective critical AFRs.

When a disk in Tiger fails, the EVManager is notified. This triggers a lookup in the EVDirectory for eclectic stripes whose chunks need to be reconstructed. The EVManager forwards the list of chunks to the ESReorganizer. For each stripe, the ESReorganizer asks the ESAllocator for disks to replace the failed disks, providing the critical AFR for the stripe. The ESAllocator returns suitable disks, if they are found, otherwise, it allocates (one or more) new eclectic stripes and moves the prior stripe's data (including any reconstructed data) to the new stripes. Finding sufficiently reliable disks to store the reconstructed data results in lower transition IO than allocating new eclectic stripes, since the latter involves moving data of disks that did not fail. After the reconstruction process (whether or not new eclectic stripes are formed), ESReorganizer informs the EVManager of the changes, which then updates the EVDirectory accordingly.

Dealing with AFR changes over time A disk's AFR is not constant throughout its lifetime [9, 10, 23, 56]. In addition to building and maintaining eclectic stripes, ESManager must also ensure that data is kept safe when a disk's AFR changes.

Ensuring data reliability with increasing AFRs. The EVManager monitors AFR by querying the change point detector.

[*]Tiger reuses the Ruptures change-point detection library [47, 48], the AFR curve-learner and the rate-limiter from HeART [25] and Pacemaker [24].

Whenever the AFR rises, the EVManager identifies any eclectic volumes whose data is at a risk of becoming under-reliable. It alerts the ESReorganizer, with the necessary stripe metadata of such stripes, which calls the ESAllocator with the current and previous disk AFR values and the number of chunks that need reallocation onto safer disks.

As with failed data reconstruction, ESAllocator prefers finding suitable disk alternates whose AFRs are less than or equal to previous AFRs values of the disks whose AFRs rose. If ESAllocator cannot find suitable disks, new eclectic stripes are formed and data is moved, as described previously.

Reducing data over-protection with reducing AFRs. When a disk's AFR decreases, there is no reliability threat to the data stored on that disk, but there may be an opportunity to reduce redundancy and obtain space-savings.

The simplest way (that also entails no transition IO cost) of reducing a stripe's redundancy is by deleting excess parities[*]. However, deleting parities is rarely an option for two reasons. First, most storage clusters have a minimum requirement on the number of parities per stripe, set by the system administrator. Second, adding/deleting a parity has a much higher impact on the MTTDL value of a stripe than adding/deleting a data chunk— deleting even a single parity usually makes the stripe miss the target MTTDL. When ESReorganizer receives metadata of possibly over-redundant stripes from the EVManager, it queries the ESMTTDLEngine whether reducing parities is feasible and, if so, enacts the change.

When deleting parities is not an option, there are two additional ways redundancy can be reduced. First, the ESAllocator could find candidate disks with AFR higher than the current disk's AFR, but low enough that the mean AFR is below the stripe's critical AFR. This method is cost-effective, since it involves only reading and writing those chunks that are on over-protected disks. Second, if the ESAllocator cannot find suitable disks, it performs new stripe allocations if it can find a new eclectic stripe with lower storage overhead. Although re-allocation has a high IO overhead (since it involves copying data over to the new stripe), it is not urgent when lowering redundancy and can be throttled by the rate limiter without putting any data at risk.

The eclectic stripe reorganizer (ESReorganizer). The ESReorganizer uses several techniques to ensure adequate reliability and provide maximum space-savings.

At any given time, the ESReorganizer might be dealing with multiple eclectic stripes seeking possible changes. ESReorganizer processes requests in priority of maintaining reliability: failed data reconstruction, then near-risk stripes that need to increase their redundancy, then requests of decommissioning disks to move data off of them, and then stripes seeking a redundancy reduction. It processes eclectic stripes that are requesting reduction in redundancy in descending order of their storage overhead.

[*]Deleting parities may not work reducing redundancy of non-MDS codes.

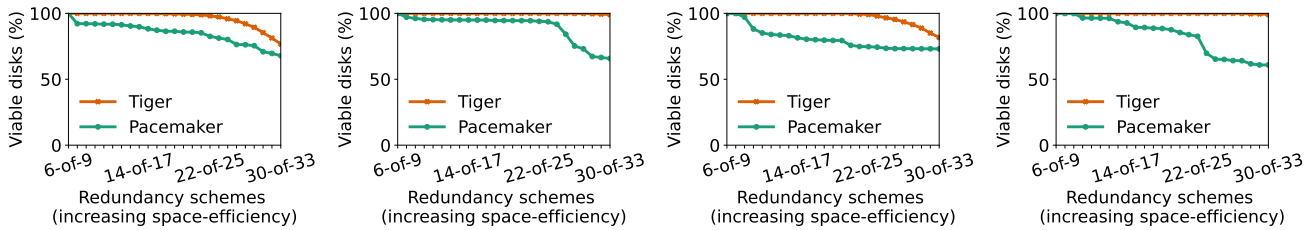
5.3 The Eclectic Volume Manager

The EVManager is responsible for creating, maintaining and monitoring the health of eclectic volumes. Recall (from §4.3) that an eclectic volume (typically in TBs) contains hundreds-of-thousands of eclectic stripes (typically in MBs). Along with health, the EVManager maintains usage statistics (e.g., freespace and load) for each eclectic volume.

Constructing and populating eclectic volumes. Similar to how ESManager manages eclectic stripes, EVManager dynamically creates and destroys eclectic volumes. The construction of the first eclectic stripe forces the creation of the first eclectic volume on the same set of disks that are chosen by the ESAllocator. When creating subsequent eclectic stripes, the ESAllocator first queries the EVManager to check if there are eclectic volumes that are conducive for storing new stripes. The EVManager does this by maintaining capacity and load-balancing metrics for each eclectic volume. Thus, the EVManager also avoids hot-spotting within eclectic volumes by spreading hot data evenly across multiple eclectic volumes. Once the target eclectic volume is identified, the set of disks comprising the eclectic volume are returned to the ESAllocator. If there is no space available, the ESAllocator gets a new set of disks from the placement policy which causes EVManager to create a new eclectic volume atop those disks. Tiger's eclectic volumes operate similar to Ceph's placement groups [51].

The Eclectic Volume Directory. Recall from §4.3 that eclectic volumes are simply a logical grouping of all the eclectic stripes with the same redundancy scheme on the same set of disks. Each eclectic volume has a unique entry in the EVDirectory and stored against the eclectic volume ID are the disks on which the eclectic volume is stored. In addition, the EVDirectory also contains a mapping from disk serial number to list of volume IDs whose fragments are stored on that disk. Note that the size of this metadata is very small. With TB-sized volume fragments, even a 100K disk storage cluster with 20TB disks will have an EVDirectory less than 100MB.

The tiny size of the EVDirectory also implies that it is unlikely to be a bottleneck. The EVDirectory will typically be queried and updated whenever disks fail, or their AFR increases significantly (in order to fetch the eclectic volumes IDs stored on the affected disks). It might also be queried to fulfill an allocation request in order to get the disks on which an eclectic volume is stored, if the eclectic-volume-to-disks mapping is not cached. Even a cluster with 500K disks has at most a few hundred disk failures in a day and typically not more than 10 makes/models, thus limiting the EVDirectory updates to less than 1000 per day. Although allocations are more frequent, caching can filter most queries for them, and their rate is also much lower than the rate of file metadata lookups in a cluster with billions of files. And, if necessary, traditional metadata scaling techniques can be employed to prevent EVDirectory from becoming a bottleneck.



(a) Backblaze **(b)** Google Cluster 1 **(c)** Google Cluster 2 **(d)** Google Cluster 3

Figure 7: Placement constraints posed by Tiger compared to Pacemaker by observing the percentage of the disk fleet that is viable for the different redundancy schemes. Tiger has lower placement constraints than Pacemaker. Tiger has over >75% disks being viable for all four clusters for all scheme configurations. Pacemaker's placement constraints are more pronounced in Google clusters since they are mostly step-deployed. This results in strict Rgroup boundaries disallowing disks from different makes/models being a part of the same Rgroup.

Reacting to failures and AFR changes. The EVHInspector continuously polls the change point detector and the cluster metadata service to gather information about disk failures and significant AFR changes. For all significant changes, the EVHInspector reconfirms the MTTDL of the affected volumes by querying the ESMTTDLEngine with the changed AFRs. Even though it is technically not a stripe, a EVDirectory has all information required to calculate the reliability of an eclectic volume, viz. the AFRs of the disks on which the volume resides, and the redundancy scheme configuration. Due to its small metadata footprint, EVHInspector can check the health of billions of stripes by checking the reliability of only thousands of eclectic volumes.

Whenever a disk fails, or a disk's AFR increases, the EVHInspector looks up the EVDirectory to find the volumes affected due to this failure / AFR rise. If the disk in question is alive, the volume manager queries the disk to obtain the stripe IDs belonging to that volume ID. If the disk has failed, the EVHInspector queries other disks of that particular eclectic volume and gathers the stripe IDs from them. Note that all disks storing a particular eclectic volume have the same list of eclectic stripe IDs in common (but they also each may have other stripes as well from non-overlapping eclectic volumes).

The EVHInspector then forwards the list of stripe IDs to the ESReorganizer along with the updated and previous AFR information and the action to be taken (reconstruct data, increase redundancy or reduce redundancy). On performing the appropriate task, the ESReorganizer communicates the metadata changes back to the EVManager, and the EVManager subsequently reflects it in the EVDirectory. For reconstruction and increase in redundancy, if a replacement disk is found, and has enough capacity to accommodate all chunks of the failed disk / disks whose AFR has increased, the eclectic volume of all constituting eclectic stripes after the operation remains the same. For redundancy reductions, or in case of not finding a replacement disk, or not finding one with enough capacity, the eclectic stripes depart from their original eclectic volume (unlike Ceph's placement groups) since they will now be stored on potentially different subset of disks.

6 Evaluation of Tiger

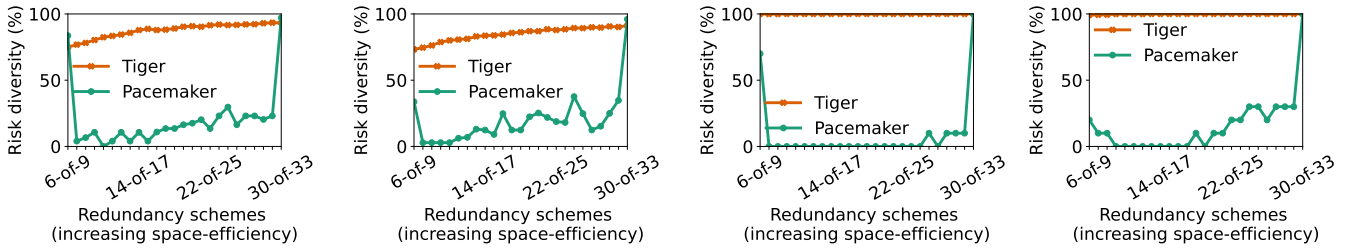
We now evaluate how Tiger performs on real-world data, and show how it fulfills the challenges laid out in §3. Tiger is evaluated using real-world deployment and failure logs from four production clusters at two different organizations (Google and Backblaze). Each cluster has a multi-year lifetime and disks from multiple makes/models/batches. Backblaze uses *trickle*-deployed disks. These disks are added to the cluster every few days in the tens or hundreds. Google Cluster 2 and Cluster 3 have *step*-deployed makes/models where disks are introduced into the cluster in large batches of tens-of-thousands of disks within a very short span of time. Google Cluster 1 is a mix of step- and trickle-deployed disks.

The highlights of our evaluation are (1) Tiger significantly lowers placement restrictions posed by Pacemaker (existing state-of-the-art disk-adaptive redundancy system); (2) Tiger's eclectic stripes provide much higher risk-diversity compared to Pacemaker; (3) Tiger is closer to the target MTTDL, and thus more efficient than existing disk-adaptive redundancy approaches; (4) Tiger outperforms Pacemaker in space-savings while keeping the average transition IO $<= 0.5\%$ and peak transition IO $< 5\%$ of cluster IO bandwidth and (5) Tiger's eclectic stripes are less sensitive to rising AFR and provide better data safety.

6.1 Tiger enables flexible data placement

We capture the flexibility in data placement by measuring the percentage of the disk fleet that is considered viable for storing data using a particular redundancy scheme. The viability is decided by whether the data stored on those disks will meet the target MTTDL. The X-axis in Fig. 7a shows the various schemes that can be supported in each storage cluster[*]. For estimating Tiger's viable disk candidates, we perform a Monte-Carlo simulation on specific days in each

[*]The narrowest scheme is set to 6-of-9 and widest is set to 30-of-33. Schemes with higher width have lower redundancy since the number of parities are kept the same. This is based on reference to prior work [24, 25], and also on the basis of communication with storage administrators of large-scale cluster storage systems at various organizations.



(a) Backblaze risk-diversity **(b)** Google Cluster 1 risk-diversity **(c)** Google Cluster 2 risk-diversity **(d)** Google Cluster 3 risk-diversity

Figure 8: Risk-diversity achieved by Tiger over three large-scale cluster storage systems. All three plots are average risk-diversity measurements taken over 5 days spread equally over the lifetime of the clusters. Pacemaker due its Rgroup based design has much lower risk-diversity compared to Tiger, more evident in Fig. 8c and 8d which are entirely step-deployed clusters.

of the cluster's lifetime. We allocate 1000 eclectic stripes by picking disks uniformly at random and check how many of the possible schemes can use the chosen disks. For Pacemaker, we bin the disks by AFRs to mimic Rgroups and measure the ratio of the population of the Rgroups to the entire disk fleet.

Tiger has almost all disks available for allocation for any scheme in Google Clusters 1 and 3 (Figs. 7b, 7d), whereas in Backblaze and Google Cluster 2 (Figs. 7a, 7c) at most 25% disks are deemed not viable for the widest schemes (beyond 22-of-25). When a large fraction of disks of the cluster have a high AFR (as is the case with Backblaze and Google Cluster 2 for the chosen dates), formation of eclectic stripes ends up with mostly high AFR disks. In such situations, Tiger cannot employ a very space-efficient redundancy scheme. Pacemaker's strict Rgroup boundaries, on the other hand, limit all disks in an Rgroup to a single scheme that may not be very wide. Therefore, for Pacemaker, all clusters see a significant drop in viable disks as the width increases.

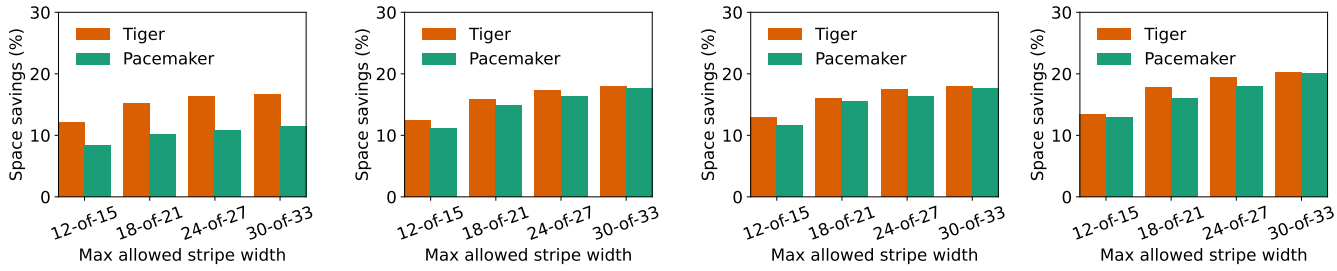
6.2 Tiger achieves high risk-diversity

Risk-diversity of a stripe is directly proportional to the number of unique makes/models participating in that stripe. If all makes/models in the storage cluster have representation in the stripe, its risk-diversity is defined to be 100%. A 0% risk-diversity implies that there were no disks in the cluster that could be used for the particular scheme. The setup used for evaluating risk-diversity is a Monte-Carlo simulation, where 100 stripes were allocated for each scheme configuration by choosing disks uniformly at random. For Tiger, we measure risk-diversity by capturing the average number of unique disk makes/models on which the chunks of an eclectic stripe are stored for each stripe configuration. For Pacemaker, we again bin the disks by AFR to form Rgroups, and count the unique number of makes/models within each Rgroup. We take the average of this simulation performed on five equally spaced days in the cluster lifetime to get an overall sense of risk-diversity of both systems.

Tiger significantly outperforms Pacemaker in providing high risk-diversity. Fig. 8 captures the risk-diversity achieved by Tiger vs Pacemaker. Since Tiger has no partitioning of disks, all disks of any make/model are viable for allocating any scheme. The minimum risk-diversity achieved by Tiger is 60% across all four clusters, that too for the narrowest scheme (6-of-9) for Backblaze (Fig. 8a) and Google Cluster 1 (Fig. 8b) clusters. Both these clusters have seven makes/models, and it is unlikely that seven out of nine chunks will be across different makes/models. As the stripe width increases, Tiger's risk-diversity also improves. Entirely step-deployed clusters, Google Cluster 2 (Fig. 8c) and Google Cluster 3 (Fig. 8d) have four and three makes/models respectively. Tiger achieves perfect risk-diversity for all possible schemes in those clusters. For Pacemaker, it is more likely that clusters where all makes/models are trickle-deployed will have a better risk-diversity because multiple makes/models can be a part of the same Rgroup so long as their AFRs are in the same range, for e.g. Backblaze (Fig. 8a). Nevertheless, even clusters with all trickle-deployed disks do not see perfect (or even good) risk-diversity since different makes/models are deployed at different times, and they go through different phases of life at different dates. Risk-diversity is poorer for Pacemaker in clusters with step-deployed makes /models as seen in Figs. 8c and 8d. This is because Rgroups and steps have a 1:1 mapping and each step only contains disks of a single make/model. The reason Pacemaker has 100% risk-diversity for 30-of-33 is because when averaging over multiple days (5 for this experiment), all makes/models on some date belonged to an Rgroup with the 30-of-33 redundancy scheme.

6.3 Tiger adapts redundancy efficiently

The efficacy of disk-adaptive redundancy performed by Tiger is evaluated using three metrics. First, we discuss the MTTDL distribution of data stored using Tiger. Subsequently, using the same four clusters used by Pacemaker we evaluate the resulting space-savings obtained by Tiger because of disk-adaptive redundancy, and finally we measure the IO overhead needed to perform necessary redundancy transitions. For fair comparison, when evaluating Tiger, we employ the same configurations (such as the IO constraints and permitted redundancy schemes) and tools (such as the AFR curve learner and the change-point detector) that are used in Pacemaker.



(a) Backblaze space-savings **(b)** Google Cluster 1 space-savings **(c)** Google Cluster 2 space-savings **(d)** Google Cluster 3 space-savings

Figure 9: Space-savings achieved by Tiger for disk-adaptive redundancy simulated on four production clusters compared to Pacemaker over conventional one-scheme-fits-all redundancy approaches. Figs. 9a–9d show that across all clusters with different maximum stripe width configurations, Tiger provides up to 5% higher average space-savings compared to Pacemaker.

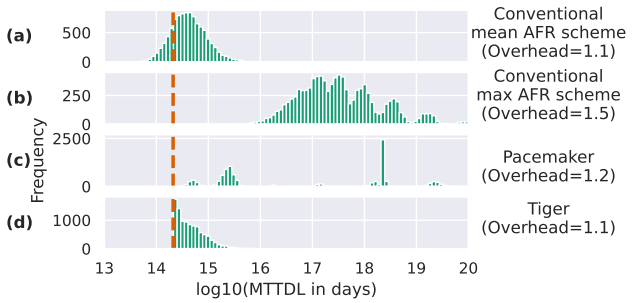


Figure 10: Comparison of MTTDL distributions for different approaches. We form 10000 random stripes for each approach using the AFRs from Google Cluster 1 (notice the different scales in the Y-axis). In a conventional system, a single scheme is chosen for all stripes based on the average AFR (*a*) or maximum AFR (*b*). (*c*) In Pacemaker, stripes must reside within an Rgroup, and the scheme depends on the Rgroup. (*d*) In Tiger, the scheme for each stripe is chosen based on the AFRs in the stripe. The dashed vertical line denotes the target MTTDL.

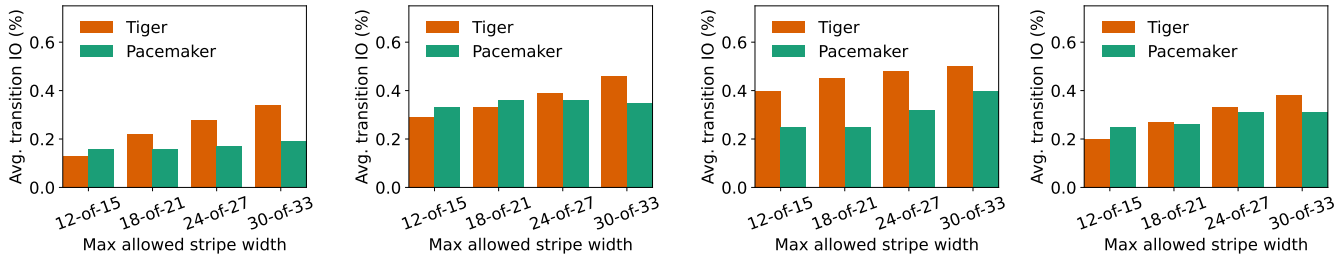
Tiger's achieves tight reliability. Storage clusters have to ensure that all data in the cluster always meets a specified target level of reliability typically specified as a MTTDL value. Tiger's target MTTDL is set as the lowest acceptable MTTDL in the system. This is calculated using the MTTDL of the most conservative homogeneous stripe possible (6-of-9) having the maximum possible AFR (16%). These settings are borrowed from Pacemaker's evaluation for a fair comparison with Tiger.

Fig. 10 shows a comparison in the distribution of stripe MTTDL with different approaches to redundancy selection for a specific day in Google Cluster 1. Fig. 10(a) shows conventional systems choosing the redundancy scheme based on the avg. AFR, which results in small storage overhead, but puts a big fraction of the stripes at risk. Fig. 10(b) shows conventional systems that choose the redundancy scheme on the basis of max AFR. Although all stripes are sufficiently protected, the storage overhead is the highest among all four alternatives. Fig. 10(c) shows Pacemaker where the different MTTDL clusters represent different Rgroups with different redundancy schemes. Pacemaker achieves good reduction in storage overhead, and keeps all stripes above the target MTTDL. In fact, some Rgroups (with higher MTTDL values) are too over-protected and denote lost opportunities for space-savings. Finally, Fig. 10(d) shows Tiger's MTTDL distribution. Despite all its eclectic stripes being above the MTTDL threshold, Tiger has least storage overhead.

Tiger achieves attractive space-savings. Akin to Pacemaker, by dynamically tailoring redundancy to disk AFRs, Tiger's eclectic stripes can use more space-efficient redundancy schemes to meet the required MTTDL target. Fig. 9 shows that Tiger achieves equal or better average space-savings compared to Pacemaker in all four clusters. For Google Clusters 1, 2 and 3 (Figs. 9b, 9c, 9d), the highly cost-efficient redundancy transitions of Pacemaker allows a large step-deployed make/model to spend more time in lower redundancy. This boosts Pacemaker's overall space-savings for these clusters and prevents Tiger from surpassing it easily.

In the Backblaze cluster (Figs. 9a), the reason for Tiger achieving better space-savings is because eclectic stripes allow high AFR disks to be mixed with low AFR disks and yet use an optimized redundancy scheme. In Pacemaker, high AFR disks cannot be mixed with other disks, resulting in lower space-savings. In the Backblaze cluster, all the seven makes/models are trickle-deployed. This results in a non-trivial fraction of disks constantly being in high-AFR regimes of infancy or wearout. While Pacemaker is forced to use the default, most conservative redundancy scheme on these disks, Tiger can use these disks for more space-efficient redundancy schemes by combining them with other, more robust disks. As a result, Tiger is able to achieve up to 5% higher space-savings compared to Pacemaker.

Tiger has very low IO overhead. Fig. 11 shows the IO overhead comparison between Pacemaker and Tiger. Although both systems are capped at 5% and in general require very low IO (compared to background tasks such as scrubbing that requires $\approx 7\%$ [5]), our evaluation shows that Tiger can achieve all its benefits with an average IO bandwidth required for redundancy transitions of at most 0.5%. In an absolute sense, Tiger's low IO overhead is mainly attributed to Tiger's efficient redundancy transitions for an AFR rise (detailed in §5.2), where Tiger moves the potentially risky chunk from



(a) Backblaze IO overhead **(b)** Google Cluster 1 IO overhead **(c)** Google Cluster 2 IO overhead **(d)** Google Cluster 3 IO overhead

Figure 11: IO overhead of redundancy scheme transitions of Tiger versus Pacemaker. In most configurations, Tiger has a higher IO overhead compared to Pacemaker due to Pacemaker leveraging its IO-efficient transitioning mechanisms. Despite being higher, the average IO overhead of Tiger is still at most 0.5% of the overall cluster's IO bandwidth; much lower than existing background tasks such as scrubbing, that require approximately 7% IO bandwidth [5]

an unsafe disk to a safe disk rather than re-encoding it or reallocating it; both having a significantly higher IO cost.

Compared to Pacemaker, Tiger still incurs slightly higher IO overhead. This is due to Tiger's mechanism of coalescing space-inefficient (high-redundancy) eclectic stripes into new space-efficient (low-redundancy) eclectic stripes in response to AFR reduction by moving all chunks. It leads to more data movement compared to moving just the chunks of the high-AFR disks (as is the case when AFR rises). This is a conscious design choice made in Tiger in order to maximize space-savings for non-urgent redundancy transitions at the expense of a minor increase in the IO overhead. Moreover, Pacemaker's IO-efficient redundancy transitioning mechanisms (that are more suitable for its Rgroup-based design) further help in reducing its IO overhead.

Tiger does not experience urgent IO bursts. In order to understand the burstiness of the IO that can be experienced by Tiger compared to Pacemaker, we artificially increase the AFR of a make/model and measure the resulting transition IO load for maintaining data reliability. Fig. 12 shows the comparison of IO loads experienced by Pacemaker vs Tiger for three instances of increasing AFR of a single step-deployed make/model. Performed on three different dates in two Google clusters (Cluster 1 and Cluster 2), we observe that Pacemaker needs orders of magnitude higher IO bandwidth than Tiger to achieve the required transitions. In fact for Google Cluster 2, in both instances none of Tiger's stripes needed transitioning despite observing a 1% rise in AFR.

We explain Pacemaker's high IO requirement with an example. Suppose a 20TB disk, which can perform 100MB/s needs to transition away from using a 30-of-33 scheme. Despite using Pacemaker's optimized Type 2 transitions*, simply reading the data to recalculate new parities would require 196% of the disk's possible IO bandwidth in a day (assuming 90% fullness to match Pacemaker's setup). In a step-deployed make/model all disks of an Rgroup transition together. In order to spread out the resulting IO burst over time, Pacemaker relies on predicting the AFR rise well in advance. To maintain a 5% IO cap, Pacemaker would need to know the AFR rise at least 40 days in advance. Long-term AFR predictions are both non-robust and non-trivial.

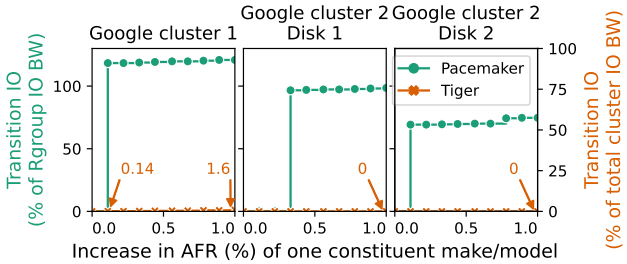


Figure 12: IO cost of redundancy transitions associated with the increase of AFR for one constituent make/model. IO cost is measured as a percentage of the total IO bandwidth of the Rgroup for Pacemaker, whereas it is the total cluster IO bandwidth for Tiger. It is calculated by scaling up a simulation of 1000 random stripes in each system and measuring the number of stripes that become unsafe after the given increase in AFR.

In contrast, Tiger for the same transition does not suffer from any IO bursts. Firstly, because of eclectic stripes, even if the disk AFR increases, only a limited fraction of data stored on it will need a redundancy transition, since other stripes might be residing on more robust disks and might continue to meet the target MTTDL. Secondly, other disks over which the eclectic stripes needing an increase in redundancy are spread need not (and probably will not) belong to the same make/model/batch. Therefore, they will not require a simultaneous increase in redundancy and can assist in transitioning data from the affected stripes. Thus disks in Tiger are spared from any sudden IO bursts.

6.4 Challenging situations for Tiger

There are certain situations that create fundamental challenges for Tiger and other disk-adaptive redundancy systems.

Sudden rise in AFRs mimicking bulk failures. Although

*In Type 2 transitions, Pacemaker re-encodes data from one scheme to another without re-writing any data. It simply recalculates new parities, writes them and deletes the old ones.

Fig. 12 shows that Tiger is robust to AFR rises in any make/model in a cluster, there could be bulk failure scenarios where large fraction of the disks in the cluster fail together. On such occasions, any system (including Tiger) that depends on redundancy will suffer from potential data loss unless the system includes cross-cluster redundancy.

A cluster with a single step-deployed make/model. Suppose a cluster had only one make/model, deployed in a step-deployed manner (note: we have not come across such an example for the large clusters targeted): there would be no diversity to exploit and all disks of the cluster would undergo redundancy transitions together. Not only would this produce bursty IO, but also will potentially result in a capacity crunch (when increasing redundancy). Such clusters would either need to keep some space unutilized to account for the bulk redundancy-increasing transitions, or will need to make provisions to add more disks to the cluster before the redundancy-increasing transitions are issued.

7 Additional Related Work

The closest related works, HeART and Pacemaker, are discussed in §2 together with other background. Additional related works can be divided into works that study the reliability of disks and distributed storage, and systems that manage multiple EC schemes and transitions between them. One essential part of disk-adaptive redundancy is the monitoring of disk AFRs, which are used by Tiger to assess the reliability of stripes. Many works have studied the behavior of disk AFRs and their impact on distributed storage reliability [5, 8, 18, 22, 26, 34, 35, 41–44]. Also, multiple works have studied the prediction of disk AFRs based on different features [1, 17, 27, 32, 45, 49, 59].

Many existing distributed storage systems allow for multiple EC schemes to coexist in the same cluster [11, 14]. There are systems that propose choosing different EC schemes for different data [46, 55]. The problem of transitioning data from one EC scheme to another has been widely studied in the Coding Theory literature, with many works studying its cost, as well as proposing special code designs that reduce the cost of transitions [20, 28–31, 36, 38, 39, 53–55, 57, 60]. Such designs could be used with Tiger, though our evaluations indicate that transition IO is not a significant problem.

8 Conclusion

Tiger enables disk-adaptive redundancy without the placement restrictions and associated problems that plague prior designs. Tiger's eclectic stripes tailor redundancy to whichever disks are chosen for each stripe. Our evaluations indicate that it reduces risk in two major ways: by increasing disk-type diversity in stripes and by reducing burstiness of transition IO urgency. Taken together, Tiger makes disk-adaptive redundancy practical for adoption in real storage clusters.

9 Acknowledgements

We thank our shepherd Gala Yadgar and the anonymous reviewers for their valuable feedback and suggestions. We extend special thanks to Larry Greenfield and numerous other researchers and engineers at Google. This research is supported in part by NSF grants CNS1956271 and CNS1901410. We also thank the members and companies of the PDL Consortium (Amazon, Google, HPE, Hitachi, IBM, Intel, Meta, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung, Seagate, Two Sigma, Western Digital) and VMware for their interest, insights, feedback, and support.

A Derivation of approximation of MTTDL of eclectic stripes

In order to approximate the MTTDL of an eclectic stripe, we will assume that the stripe can be repaired in the data loss state and we will approximate the MTTDL as the mean time between visits to the data loss state. In particular, we will analyze the stripe as an alternating renewal process. Let A_s be the stripe availability (i.e., the fraction of the time that the stripe is not in the data loss state), μ_s be the repair rate in the data loss state, and λ_s the stripe data loss rate. As described above, the MTTDL is approximately λ_s^{-1}. For an alternating renewal process, we have that:

$$A_s = \frac{\mu_s}{\mu_s + \lambda_s} \iff \frac{1}{\lambda_s} = \frac{A_s}{\mu_s(1-A_s)} \tag{2}$$

The repair rate in the data loss state is simply the number of failed disks in that state:

$$\mu_s = (n-k+1)\mu. \tag{3}$$

We assume that each disk in the stripe fails independently from the rest, and that it is repaired with rate μ if it fails. Then, in steady state, disk i is available with probability:

$$A_i = \frac{\mu}{\mu+\lambda_i}. \tag{4}$$

Let $P(j)$ be the probability that we find the stripe in a state where exactly j disks are available in the stripe. Since there are no states with more than $n-k+1$ failed disks, we have that:

$$P(j) = \frac{Q(j)}{Q(k-1)+\cdots+Q(n)}, \text{for } k-1 \le j \le n, \tag{5}$$

where $Q(j)$ is the probability that exactly j disks are available. Since disks are independent, $Q(j)$ is equal to a Poisson-binomial distribution, with probabilities $(A_i)_{i=1}^n$. Given this,

the availability of stripe is given by:

$$A_s = P(k) + \cdots + P(n). \tag{6}$$

Thus, we have:

$$\frac{1}{\lambda_s} = \frac{Q(k) + \cdots + Q(n)}{\mu(n-k+1)Q(k-1)} \approx \frac{1}{\mu(n-k+1)Q(k-1)}. \tag{7}$$

Where the approximation comes from the fact that $Q(n) \approx 1$ because $\mu \gg \max_i \lambda_i$ and thus all A_i are close to 1.

In summary, we have that:

$$\text{MTTDL} \approx \frac{1}{\mu(n-k+1)Q(k-1)}. \tag{8}$$

References

[1] Preethi Anantharaman, Mu Qiao, and Divyesh Jadav. Large Scale Predictive Analytics for Hard Disk Remaining Useful Life Estimation. In *IEEE International Conference on Big Data*, 2018.

[2] John E Angus. On computing MTBF for a k-out-of-n: G repairable system. *IEEE Transactions on Reliability*, 37(3):312–313, 1988.

[3] Backblaze. Disk Reliability Dataset. https://www.backblaze.com/b2/hard-drive-test-data.html, 2013-2018.

[4] Backblaze. Erasure coding used by Backblaze. https://www.backblaze.com/blog/reed-solomon/, 2013-2018.

[5] Lakshmi N Bairavasundaram, Garth R Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[6] Werner Ehm. Binomial approximation to the Poisson binomial distribution. *Statistics & Probability Letters*, 11(1):7–16, 1991.

[7] Nosayba El-Sayed, Ioan A Stefanovici, George Amvrosiadis, Andy A Hwang, and Bianca Schroeder. Temperature management in data centers: Why some (might) like it hot. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 163–174, 2012.

[8] Jon Elerath. Hard-disk drives: The good, the bad, and the ugly. *Communication of ACM*, 2009.

[9] Jon G Elerath. AFR: problems of definition, calculation and measurement in a commercial environment. In *IEEE Reliability and Maintenance Symposium (RAMS)*, 2000.

[10] Jon G Elerath. Specifying reliability in the disk drive industry: No more MTBF's. In *IEEE Reliability and Maintenance Symposium (RAMS)*, 2000.

[11] Erasure code Ceph Documentation. https://docs.ceph.com/docs/master/rados/operations/erasure-code/, (accessed September 25, 2019).

[12] Manuel Fernández and Stuart Williams. Closed-form expression for the Poisson-binomial probability density function. *IEEE Transactions on Aerospace and Electronic Systems*, 46(2):803–817, 2010.

[13] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[14] Apache Software Foundation. HDFS Erasure Coding. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2017 (accessed November 5, 2020).

[15] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, 2003.

[16] Garth Alan Gibson. *Redundant disk arrays: Reliable, parallel secondary storage*. PhD thesis, University of California, Berkeley, 1991.

[17] Greg Hamerly, Charles Elkan, et al. Bayesian approaches to failure prediction for disk drives. In *International Conference on Machine Learning (ICML)*, 2001.

[18] Eric Heien, Derrick Kondo, Ana Gainaru, Dan LaPine, Bill Kramer, and Franck Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *ACM / IEEE High Performance Computing Networking, Storage and Analysis (SC)*, 2011.

[19] Yili Hong. On computing the distribution function for the Poisson binomial distribution. *Computational Statistics & Data Analysis*, 59:41–51, 2013.

[20] Yuchong Hu, Xiaoyang Zhang, Patrick P. C. Lee, and Pan Zhou. Generalized optimal storage scaling via network coding. In *IEEE International Symposium on Information Theory (ISIT)*, 2018.

[21] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)*, 2012.

[22] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *ACM Transactions on Storage (TOS)*, 2008.

[23] Saurabh Kadekodi. *DISK-ADAPTIVE REDUNDANCY: tailoring data redundancy to disk-reliability heterogeneity in cluster storage systems*. PhD thesis, Carnegie Mellon University, 2020.

[24] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, KV Rashmi, and Gregory R Ganger. PACEMAKER: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[25] Saurabh Kadekodi, K V Rashmi, and Gregory R Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *USENIX File and Storage Technologies (FAST)*, 2019.

[26] Ao Ma, Rachel Traylor, Fred Douglis, Mark Chamness, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. RAIDShield: characterizing, monitoring, and proactively protecting against disk failures. *ACM Transactions on Storage (TOS)*, 2015.

[27] Farzaneh Mahdisoltani, Ioan Stefanovici, and Bianca Schroeder. Proactive error prediction to improve storage system reliability. In *USENIX Annual Technical Conference (ATC)*, 2017.

[28] Francisco Maturana, V. S. Chaitanya Mukka, and K. V. Rashmi. Access-optimal linear MDS convertible codes for all parameters. In *IEEE International Symposium on Information Theory (ISIT)*, 2020.

[29] Francisco Maturana and K. V. Rashmi. Bandwidth cost of code conversions in distributed storage: Fundamental limits and optimal constructions. *arXiv preprint arXiv:2008.12707*, 2020.

[30] Francisco Maturana and K. V. Rashmi. Convertible codes: new class of codes for efficient conversion of coded data in distributed storage. In *Innovations in Theoretical Computer Science Conference, (ITCS)*, 2020.

[31] Sara Mousavi, Tianli Zhou, and Chao Tian. Delayed parity generation in MDS storage codes. In *IEEE International Symposium on Information Theory (ISIT)*, 2018.

[32] Joseph F Murray, Gordon F Hughes, and Kenneth Kreutz-Delgado. Hard drive failure prediction using non-parametric statistical methods. In *Springer Artificial Neural Networks and Neural Information Processing (ICANN/CONIP*, 2003.

[33] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pages 217–231, 2021.

[34] David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM International Conference on Management of Data (SIGMOD)*, 1988.

[35] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *USENIX File and Storage Technologies (FAST)*, 2007.

[36] Brijesh Kumar Rai, Vommi Dhoorjati, Lokesh Saini, and Amit K. Jha. On adaptive distributed storage systems. In *IEEE International Symposium on Information Theory (ISIT)*, 2015.

[37] K V Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[38] K. V. Rashmi, Nihar B. Shah, and P. Vijay Kumar. Enabling node repair in any erasure code for distributed storage. In *IEEE International Symposium on Information Theory (ISIT)*, 2011.

[39] KV Rashmi, Nihar B Shah, and Kannan Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. *IEEE Transactions on Information Theory*, 2017.

[40] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. In *International Conference on Very Large Data Bases (VLDB)*, 2013.

[41] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding latent sector errors and how to protect against them. *ACM Transactions on Storage (TOS)*, 2010.

[42] Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *USENIX File and Storage Technologies (FAST)*, 2007.

[43] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*. IOP Publishing, 2007.

[44] Sandeep Shah and Jon G Elerath. Disk drive vintage and its effect on reliability. In *IEEE Reliability and Maintenance Symposium (RAMS)*, 2004.

[45] Brian D Strom, SungChang Lee, George W Tyndall, and Andrei Khurshudov. Hard disk drive reliability modeling and failure prediction. *IEEE Transactions on Magnetics*, 2007.

[46] Eno Thereska, Michael Abd-El-Malek, Jay J Wylie, Dushyanth Narayanan, and Gregory R Ganger. Informed data distribution selection in a self-predicting storage system. In *IEEE International Conference on Autonomic Computing (ICAC)*, 2006.

[47] Charles Truong, Laurent Oudre, and Nicolas Vayatis. A review of change point detection methods. In *arXiv:1801.00718v1 [cs.CE]*, 2018.

[48] Charles Truong, Laurent Oudre, and Nicolas Vayatis. ruptures: change point detection in python. In *arXiv:1801.00826v1 [cs.CE]*, 2018.

[49] Yu Wang, Eden WM Ma, Tommy WS Chow, and Kwok-Leung Tsui. A two-step parametric method for failure prediction in hard disk drives. *IEEE Transactions on industrial informatics*, 2014.

[50] Hakim Weatherspoon and John D Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Springer International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[51] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[52] Wolfram. Wolfram Mathematica. https://www.wolfram.com/mathematica.

[53] Si Wu, Zhirong Shen, and Patrick P. C. Lee. Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic Reed-Solomon codes. In *39th Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020*, 2020.

[54] Si Wu, Yinlong Xu, Yongkun Li, and Zhijia Yang. I/O-efficient scaling schemes for distributed storage systems with CRS codes. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[55] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. A tale of two erasure codes in HDFS. In *USENIX File and Storage Technologies (FAST)*, 2015.

[56] Jimmy Yang and Feng-Bin Sun. A comprehensive review of hard-disk drive reliability. In *IEEE Reliability and Maintenance Symposium (RAMS)*, 1999.

[57] Xiaoyang Zhang, Yuchong Hu, Patrick P. C. Lee, and Pan Zhou. Toward optimal storage scaling via network coding: from theory to practice. In *IEEE Conference on Computer Communications, (INFOCOM)*, 2018.

[58] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does erasure coding have a role to play in my data center. *Microsoft research MSR-TR-2010*, 52, 2010.

[59] Ying Zhao, Xiang Liu, Siqing Gan, and Weimin Zheng. Predicting disk failures with HMM-and HSMM-based approaches. In *Springer Industrial Conference on Data Mining (ICDM)*, 2010.

[60] Weimin Zheng and Guangyan Zhang. Fastscale: accelerate RAID scaling by minimizing data migration. In *USENIX File and Storage Technologies (FAST)*, 2011.

zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO

Tim Stamler[1] Deukyeon Hwang[1] Amanda Raybuck[1] Wei Zhang[2] Simon Peter[3]
[1]UT Austin [2]Microsoft [3]University of Washington

Abstract

We present zIO, a transparent zero-copy IO mechanism for unmodified IO-intensive applications. zIO tracks IO data through the application, eliminating copies that are unnecessary while maintaining data consistency.

Applications often modify only a part of the data they process. zIO leverages this insight and interposes on IO stack and standard library memory copy calls to track IO data and eliminate unnecessary copies. Instead, intermediate data locations are unmapped, allowing zIO to intercept and resolve any access via page faults to maintain data consistency. To avoid harming application performance in situations where data tracking overhead is high, zIO's tracking policy decides on a per IO basis when to eliminate copies. Further, we demonstrate how to use zIO to achieve *optimistic network receiver persistence* for applications storing data from the network in non-volatile memory (NVM). By mapping socket receive buffers in NVM and leveraging kernel-bypass IO, we can rely on zIO to transparently eliminate all copies from the network, through the application, to storage.

We implement zIO as a user-space library. On top of kernel IO stacks, zIO eliminates application-level IO copies. We also integrate zIO with kernel-bypass IO stacks, where it can additionally eliminate copies incurred by the IO stack APIs and enable optimistic network receiver persistence. We evaluate zIO with IO-intensive applications, such as Redis, Icecast, and MongoDB. zIO improves application throughput by up to 1.8× with Linux and by up to 2.5× with kernel-bypass IO stacks and optimistic network receiver persistence. Compared to common uses of zero-copy IO stack APIs, such as memory mapped files, zIO can improve performance by up to 17% due to reduced TLB shootdown overhead.

1 Introduction

Zero-copy IO has been a long-standing performance goal. Copies introduce memory and CPU overhead, limiting the performance of IO-intensive applications. IO data copies are performed within IO stacks, by their application programming interfaces (APIs), and within applications. Existing work has focused on eliminating copies within IO stacks [27, 28] and within IO stack APIs by developing zero-copy IO APIs [1, 11, 12, 15, 17, 28, 32], including some that strive for transparency [8, 9, 22].

Despite these advances, data from IO is still copied. We find that IO-intensive applications perform up to 8 copies of request data for each IO request (cf. §2.1). Many of these copies occur among subsystems within the applications themselves (*application copies*). Only a fraction is performed at the IO stack API (*IO copies*—for example, many standard POSIX socket and file IO system calls copy data between system and user-provided buffers).

A reason for the continued adoption of copies is that they simplify development. Copies are used as a robust mechanism to pass ownership of data among independent subsystems. A data buffer local to a subsystem cannot be touched by a caller of the subsystem, allowing for subsystem-internal use of the data without worry of corruption or deallocation of the memory backing the data from the outside. For example, copies are used to simplify asynchronous IO. POSIX allows kernel IO stacks to provide internal buffers to IO devices that operate asynchronously. Applications request and copy IO data into user-space buffers, allowing applications synchronous processing of a single buffer at a time, while the IO stack recycles its internal buffers for further asynchronous IO. Finally, applications use copies to simplify data handling, for example to perform alignment, padding, serialization and deserialization, as well as bucketization (cf. §2.2).

Unfortunately, copying is an imperfect tool. While copies provide the aforementioned benefits, they also introduce overhead. Using the Redis [21] key-value store as an example IO-intensive application, we study the overhead of copies for IO, both using Linux kernel IO stacks (§2.2) and using kernel-bypass IO stacks for high-bandwidth IO devices (§2.3). IO copying overhead scales with the amount of copied IO data. As IO devices, in particular for storage and networking, increase bandwidth, copies become the performance-limiting factor in IO-intensive applications [7].

The question we ask is: can we attain the benefits of simple development offered by copying, while alleviating its increasing overheads? As we have seen, application developers opt for copies regardless of the availability of zero-copy IO APIs. We find that zero-copy APIs require application modification, increase code complexity, and are not widely supported (§2.4). Hence, we strive for a solution that allows application developers the freedom to program with copies and to use any IO API, while *transparently* eliminating copies where it makes sense, without requiring application modification.

We present zIO, a transparent zero-copy IO mechanism for IO-intensive applications. IO-intensive applications act between IO stacks, examining and potentially transforming input data before output. zIO tracks data that is read by applications from IO stacks to its final destination (typically another

IO stack, but the data may also be held in memory). In the process, zIO eliminates copies that are unnecessary while maintaining consistency for data that the application accesses. By tracking data and eliminating copies, zIO minimizes the overhead incurred by copies, improving application performance.

zIO works under the assumption that IO-intensive applications often touch only a part of the data they process. Untouched data may remain in its original place, while touched data continues to be copied. However, the challenge is that we do not know a priori what data will be touched. zIO optimistically assumes that most data will remain untouched and, by interposing on IO system calls and C standard library calls like memcpy and memmove, eliminates copies, instead simply marking the target memory area as *intermediate*. zIO can do so transitively for entire copy chains. To maintain consistency, each target area remains unmapped. If the application attempts to touch any intermediate memory area, zIO intercepts the access via a page fault. In this case, zIO performs the copy for touched pages and remaps them. Another challenge is to deal with unaligned memory areas. In this case, zIO performs the copy of unaligned sections of the area and leaves only page-aligned portions unmapped. Unaligned sections are small and copying them does not harm performance.

To avoid harming application performance due to data tracking overhead, zIO dynamically decides on a per IO basis when to track and when to copy (via its *tracking policy*). If the size of an IO buffer is smaller than 16KB, zIO copies the buffer. zIO also tracks the average number of page faults and eliminated copied bytes per buffer. If the ratio of bytes accessed to bytes eliminated from copies exceeds 6%, we impose too much overhead handling page faults to improve application performance and zIO copies the buffer instead.

In addition to eliminating application copies, we also use zIO to eliminate copies across IO stack APIs. To do so, we use kernel-bypass IO stacks in addition to zIO. Kernel-bypass stacks use shared memory to implement their APIs, allowing zIO to track IO as it arrives from the IO devices and eliminate copies, even across the IO stack API. We implement these changes in the TAS [18] network stack and the Strata [20] file system. We discuss how to apply these principles to any IO stack in (§3.4).

By leveraging non-volatile memory (NVM), we achieve a further optimization: *optimistic input persistence*. If input received from an IO stack is persisted in NVM via a storage stack by applications, optimistic input persistence enables end-to-end transparent elimination of copies through to storage. To do so without violating application data persistence requirements, we extend zIO to identify NVM mappings. Data copies to NVM may be eliminated if the original data already resides in NVM. Otherwise, a copy is necessary to enforce persistence. Using this technique, we demonstrate how to achieve *optimistic network receiver persistence* by mapping socket receive buffers in NVM and relying on zIO to transparently eliminate all copies through to the file system.

We make the following contributions:

- An analysis of copying in IO-intensive applications (§2). We study the number of copies made in popular IO-intensive applications and find that copies are common, in particular within applications themselves. We conduct a case study of copies in the Redis key-value store, analyzing when and why copies are carried out. Finally, using the Redis case study, we demonstrate that copies are a performance bottleneck for IO-intensive applications, especially when leveraging optimized kernel-bypass IO stacks.
- We present zIO, a transparent zero-copy IO system for IO-intensive applications. zIO addresses the presented overheads due to copying. We show how to use zIO to eliminate application-level copies. We show how to eliminate IO stack API copies when combining zIO with kernel-bypass IO stacks. We show how to achieve optimistic input persistence by leveraging NVM.
- We implement zIO as a user-space library. When executing on top of the Linux kernel network and storage stacks, zIO successfully eliminates application copies of IO buffers. We also integrate zIO with the kernel-bypass IO stacks TAS [18] and Strata [20], enabling it to additionally eliminate copies performed by the IO stack APIs.
- We break down zIO's performance contributions with microbenchmarks and analyze the overheads of buffer tracking. We evaluate the performance benefit to IO-intensive applications, like Redis [21], Icecast [37], and MongoDB [25] and compare to Linux and kernel-bypass IO without copy elimination, where zIO improves performance by up to 1.8× and 2.5×, respectively. We also compare zIO's performance to common uses of zero-copy IO stack APIs, such as memory mapped files, where zIO can improve performance by up to 17% due to reduced TLB shootdown overhead.

2 Background

IO-intensive applications often make several copies of IO data while processing it. We survey the prevalence of these copies in IO-intensive applications (§2.1). To learn how copies are used for IO, we study one of these applications, Redis, and investigate how it uses copies to do IO processing (§2.2). Looking forward, we investigate how copies can become a limiting factor to IO performance (§2.3). Zero-copy APIs are a potential alternative to IO copies. We study their intended use and the tradeoffs they make (§2.4).

2.1 Copies in IO-Intensive Applications

We study the prevalence of IO data copies in popular IO-intensive applications. We identify the call site of these copies and break down occurrences into copies that are involved in an IO stack API call and copies occurring within application subsystems. Our methodology involves a source code analysis of IO data flows through application subsystems from input to output. We identify what methods applications use

Application	Operation	Copy call site App	IO Stack
Redis [21]	SET	4	2
	GET	2	1
Icecast [37]	Cast to N clients	0	1 + N
Ceph [34]	Write	1	2
	Read	0	2
Anna [36]	PUT	5	3
	GET	4	3
MongoDB [25]	Insert	3	2
	Disk sync	1	1
	Read	2	2
Tensorflow-serving [26]	Inference	2	1
Nebula Graph [33]	Insert vertex	5	2
	Store a vertex	4	3

Table 1. Number and call site of copies between input and output for various application operations.

to copy IO data and how copies are affected by the executed functionality and its parameters. We find that all applications investigated use C standard library functions, such as `memcpy` and `memmove`, to copy data. We use this insight to validate our source code analysis via an execution of the relevant application operations under a debugger set to break on these C library memory copy APIs. For each application operation, we count the number of breakpoints hit on IO code paths between input and output and check that the count matches that of our source code analysis.

Table 1 presents the number of copies made at the IO stack and within various IO-intensive applications, broken down by operation. We are specifically interested in the copy of potentially large IO data, as small data copies do not significantly impact application performance. For example, the Anna [36] key-value store conducts up to 45 copies of keys during a PUT operation. We ignore these copies in the table.

While the number of copies varies among applications and operations, we can see that IO-intensive applications extensively copy IO data between input and output. We can also see that applications often make more internal copies of IO data than at the IO stack API. For example, Redis [21] makes twice as many application-internal copies than at the IO stack for a SET request. IO-intensive applications also often employ third-party libraries. For example, the Anna [36] key-value store uses gRPC [14] and Protobuf [13] to serialize and deserialize data. We observe that these libraries incur up to 3 per-IO data copies for this task, leading Anna to make up to 5 internal IO copies. This indicates that zero-copy IO APIs are only going to eliminate a fraction of the overhead due to copies. Application-internal copies, including in third-party libraries, often constitute a similar or even larger fraction of copy-induced CPU overhead.

2.2 Copy Case Study: Redis

To better understand these IO data copies, we study the Redis SET request. Redis [21] is a popular key-value store providing a rich RPC-based network API to an in-memory store, persisted via snapshots or operation logging. We configure Redis to log SET operations to study a use-case that is equally network and storage IO intensive. In our study, each SET request provides a new value, identified by a 32 byte key. We run a single-threaded Redis server instance on the evaluation platform described in Section 5. We use `redis-benchmark` [21] to attach 64 clients over a 100G network, enough to saturate the server. We configure Redis to use an append-only file to persist data without delay. This configuration provides strong crash consistency—every operation is persisted before it is acknowledged. We evaluate the number of memory copies that Redis performs per SET request and we study these copies.

#	Source	Destination	Call site
IO_1	Socket buffer	`c.socket_buf`	`readQueryFromClient`
A_1	`c.socket_buf`	`c.socket_buf`	`processInputBuffer`
A_2	`c.socket_buf`	`hash_node`	`dbAdd`
A_3	`c.socket_buf`	`c.write_to_aof`	`feedAppendOnlyFile`
A_4	`c.write_to_aof`	`aof_buf`	`flushAppendOnlyFile`
IO_2	`aof_buf`	Append-only file	`flushAppendOnlyFile`

Table 2. Copies in Redis SET request. IO_i are IO stack copies, A_j are application copies. `c` is a per-client structure.

As reported in Table 1, we find that Redis copies request data 6 times for each SET request. We list these copies and their call sites in Table 2. As we can see, Redis performs copies to read and deserialize the SET request and to store the request both in an in-memory hash table and in the append-only file. After reading a number of kilobytes from the network socket to an input buffer (copy IO_1), Redis identifies the next request within the input buffer and removes its headers from the buffer (copy A_1). If the identified SET request is admissible, Redis creates a copy of the key and value data to store in its in-memory hash table (copy A_2). Redis then reformats the request so it can be logged to its append-only file and appends the request to a per-client log (copy A_3). Redis uses per-client logs to support *group commit*—Redis can process a number of pending client requests in-memory and then persist and acknowledge these requests in a batch, eliminating storage stack overheads incurred for small IO. To do so, Redis first combines all pending per-client logs into a single log stream (copy A_4) and then writes the log stream to the append-only file (copy IO_2).

All of these copies could have been avoided. However, it would have required the Redis developers to design a complex set of coordinated, reference counted buffer descriptors that can track each request and its data in each source buffer (in this case, a network socket buffer). Reference counts provide use-after-free protection. Use-after-free [38] is an error condition where one part of an application or IO stack frees an allocated IO data buffer and re-uses it for other purposes while another part of the application or an IO device still uses the data. Use-after-free protection requires complex ownership tracking, including APIs to convey ownership transfer. Further, fine-grained memory management is required, including the ability to free fragments of a previously allocated memory

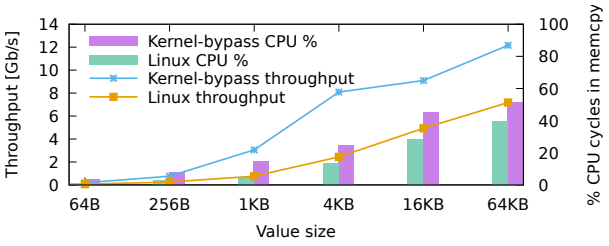


Figure 1. Redis SET throughput and fraction of CPU cycles in `memcpy` over value size, with and without kernel-bypass.

buffer. For example, the headers of incoming SET requests can be freed after each request is processed, while the keys and values remain in their original buffers for as long as they are stored in the key-value store. This creates buffer fragmentation that is difficult to resolve via memory management alone, requiring further APIs to defragment buffers over time. All of these APIs are complex and it is often impossible to support them in an application when third-party libraries or IO stacks are used that do not support the APIs.

2.3 When is IO Performance Copy-Limited?

As hardware IO bandwidth continues to increase and IO stacks become lighter-weight to keep up with increasing application demand for bandwidth, copies start to limit IO performance. In light of these trends, we investigate the performance impact of copying for Redis SET requests over increasing value sizes, while using heavy-weight in-kernel and light-weight kernel-bypass IO stacks. We use the same Redis configuration described in Section 2.2, evaluating the Linux network stack and the ext4 file system, as well as TAS [18] and Strata [20] for kernel-bypass. As we vary the value size, we measure throughput and the fraction of CPU cycles spent in data copies per request with Linux `perf`.

The results are presented in Figure 1. We can see that larger value sizes imply higher per-core throughput. Also, kernel-bypass IO improves throughput by up to 4×. This is intuitive. Kernel-bypass IO is lighter-weight than in-kernel IO, while larger IO granularity amortizes IO stack overheads. As hardware IO bandwidth continues to increase, it is likely that applications will employ larger IO sizes to leverage the available bandwidth. At the same time, IO stacks will become lighter-weight to provide the necessary performance to keep up with the increasing IO speeds.

We can also see that larger value (and thus IO) sizes cause a noticeable increase of per-request CPU cycles spent in memory copies. We already know that Redis makes 6 copies of IO data for each SET request. As value sizes increase, the amount of CPU cycles spent copying them must naturally also increase. Even moderate value sizes of 64KB cause 39% of per-request CPU cycles to be spent in memory copies using the heavy-weight Linux kernel IO stacks. The lighter-weight kernel-bypass IO causes an even larger fraction of up to 52% of per-request CPU cycles to be spent in memory copies, owing to a reduction of per-request CPU cycles spent in IO stack processing. For even larger value sizes of 512KB, CPU cycles spent in copying reaches 60%.

2.4 Limitations of Existing Zero-Copy IO APIs

Various zero-copy APIs have been proposed to limit the number of copies involved in IO-intensive applications. Zero-copy IO APIs fall into two categories. (1) Single-stack APIs, and (2) cross-stack APIs. Single-stack APIs eliminate copies for a particular IO API, such as the network sockets system call API. Cross-stack APIs eliminate copies across IO APIs. For example, across network and storage APIs. We study the tradeoffs made by each category in this section.

Single-stack APIs. Single-stack APIs provide zero-copy IO for single IO stacks. The API is specific to the IO stack and is often provided in the form of new parameters or tweaks to a familiar IO API that enable zero-copy, typically along with a set of invocation and environment requirements that have to be met by the application developer for the API to function. We describe a number of storage and networking zero-copy APIs here, including memory mapped files, Linux FreeBSD, and Solaris zero-copy networking, remote direct memory access (RDMA), and the Arrakis [28] zero-copy networking API.

Memory mapping files is one of the oldest zero-copy storage IO APIs. Applications map (parts of) files into their virtual address space, which the OS implements by loading the file into the page cache and providing direct application access to the relevant pages. Page cache entries may be directly written to disk, without further copies. More recently, applications may also map non-volatile memory (NVM) directly into virtual memory, referred to as direct access (DAX) [2]. Memory mapped files restrict some file IO. For example, memory mapped files cannot be appended to. Instead, an application developer has to determine the file size in advance and `truncate` the file to the desired length before memory mapping it. Further, the interface does not allow applications to make atomic modifications to file data without copying data to their own buffers first.

Linux provides two networking zero-copy APIs [11, 12] for TCP sockets. A zero-copy `send` will lock a given application buffer into memory and start the transmission. If transmission is not complete by the time `send` returns, the application must take care not to touch the buffer. The zero-copy mechanism will place a notification message in the error queue associated with the socket, which has to be monitored by the application. When an "error" packet appears, it can be examined to determine the status of the operation, including whether the transmission succeeded and whether it was able to run in zero-copy mode.

For zero-copy receive, Linux allows to memory map a TCP socket. If several network conditions are met, including the next incoming data chunk being page-sized and page-aligned,

the socket buffer containing the incoming chunk will be mapped into the calling process's address space, where it can be accessed directly. When the incoming data has been processed, the application calls `munmap` to release the pages and free the buffer for another incoming packet. The mechanism only works if the application developer has knowledge of exactly what each incoming packet will look like.

RDMA [30] provides zero-copy IO either by directly reading/writing remote memory or by pre-registering buffers with the network card for receive and transmit operation. Similarly, Arrakis [28] provides a zero-copy IO network socket interface that returns buffers and consumes them, rather than letting the application specify its own buffers. All of these interfaces introduce the same complexities of buffer ownership management and knowledge of network conditions. These conditions are often difficult to meet and the additional buffer management burden is cumbersome for many developers.

A limited solution proposed by SocksDirect [22] and also implemented in FreeBSD [8] and Solaris [9] to *transparently* avoid copies in the network sockets API is to simply remap the pages carrying IO data from the network stack to the application-provided buffer location, instead of copying the data. This works in cases where both buffers are page-aligned and it requires the NIC to be able to isolate packet payloads and place them into page-aligned buffers. To isolate payloads, SocksDirect requires RDMA, while Solaris requires ATM. FreeBSD supports traditional Ethernet NICs, but requires that the maximum transfer unit is configured to be greater than the hardware page size, which may be undesirable or difficult. Unfortunately, applications often misalign IO buffers, even if memory allocators return aligned memory. For example, when headers are inserted into a buffer and IO is read to a location after the header. Our investigation into Redis shows that only about 40% of IO data can be remapped using this approach. Further, transmit buffers must be kept until acknowledged, leaking memory if acknowledgments lag. The limited applicability, security concerns (including from malicious NICs [24]), and hardware requirements led the FreeBSD developers to abandon the transparent zero-copy socket API in FreeBSD 11.

Cross-stack APIs. A variety of cross-stack APIs attempt to eliminate copies across IO stacks, in particular the networking and storage stacks. To do so, they offer new and often higher-level APIs that the application developer must use. These new APIs avoid copies. We describe three example cross-stack APIs here, the Linux `sendfile` family of system calls, PASTE, and Demikernel.

The Linux `sendfile` system call (and cousins `splice` for pipes and `copy_file_range` for files) transmits data from the storage stack via the network stack without user-level copies. The API is restricted to network and storage IO and does not permit the application developer to inspect data before transmission. To add any application data, such as headers, the developer must use the `TCP_CORK` option, requiring them to add the necessary data within a 200 millisecond time window. `sendfile` does not allow sending or receiving from/to user memory. The API is used to send static files across the network but is increasingly obsolete with the prevalence of dynamic in-memory content.

PASTE [15] provides an API that combines the network stack with persistent data structures in NVM to avoid copies. PASTE builds on the Netmap [31] kernel framework to place packets from the network interface card (NIC) directly in NVM. Developers can refer to these packets from application-specific persistent data structures. However, PASTE operates at the packet level and requires developers to track network connections and decode byte streams to find relevant data to persist. PASTE also requires the developer to implement a copy-on-write scheme to efficiently return packet buffer space to the NIC after use. Due to the complexity of its API, PASTE's intended use is constrained to run-to-completion processing of requests that fit in individual network packets.

Demikernel [38] eliminates copies between kernel-bypass networking stacks, like DPDK and RDMA, and kernel-bypass storage stacks, like SPDK. The Demikernel memory manager allocates memory to applications from DPDK's memory pool and it registers that memory with RDMA. This allows Demikernel applications to receive data over the network and to store it without any copies. Demikernel offers a queue-oriented interface, PDPIX, which replaces datapath IO calls with pushes and pops to and from queues that may return tokens if data is unavailable. Demikernel's interface requires application developers to implement run-to-completion IO processing. This simplifies zero-copy IO for Demikernel, but it limits the application developer. The Demikernel interface does not support making in-place updates to IO data or allow developers to schedule input and output beyond handling each input request to completion, and it cannot eliminate any further copies an application might make internally to process input.

Summary. Both categories of zero-copy IO stacks seek to eliminate copies involved in IO stack APIs. However, in doing so they introduce complexities, such as buffer ownership management involving special API calls. They also introduce restrictions, such as requiring run-to-completion processing, buffer alignment, or disallowing in-place updates. Finally, they may enforce external IO properties, such as packet layout and MTU size. These complexities and restrictions are difficult for developers to navigate and external IO properties are often difficult or impossible to enforce. Further, **none of the existing zero-copy APIs provide transparent copy elimination across IO stacks or eliminate copies that are made within the application**. For these reasons, both application and kernel developers forgo zero-copy APIs, as they often struggle to outperform APIs that involve copies and are deemed not worth the complexity they introduce [12].

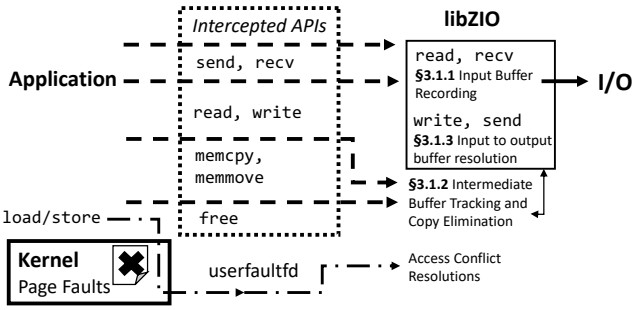


Figure 2. zIO overview.

3 zIO Design

zIO is a user-level library (libzIO) that may be dynamically and transparently linked to applications. zIO intercepts a number of C standard library and IO system calls (shown in Figure 2), including memory copy and management, and socket and file IO. zIO leverages userfaultfd [5] to intercept page faults, which may be caused by applications touching intermediate memory buffers. We now describe zIO in three parts. First, we describe how zIO tracks data within the application to eliminate *application copies* (§3.1). Second, we describe how we extend zIO with kernel-bypass IO stacks to allow it to eliminate *IO stack copies* (§3.2). Third, we describe how to realize *optimistic input persistence* by mapping appropriate IO buffers into NVM (§3.3).

3.1 Application Copy Elimination

To eliminate application copies, zIO tracks IO data buffer locations transitively through the application. zIO intercepts any copies of IO buffers and optimistically forgoes them. To provide data consistency in the face of the application accessing any intermediate, uncopied buffer locations, zIO leverages page faults to detect and resolve these accesses.

Figure 3 shows the mechanisms involved in this process via an example involving a key-value pair being read from an input IO stack, processed by the application, and written to an output IO stack. On input (e.g., IO stack `read/recv` calls), the provided location of the application buffer is recorded by zIO (①). For the purpose of application copy elimination, this is the original location of the IO data. zIO uses this information to track and eliminate any application-level copies of this data. Upon memory copy of any tracked data (`memcpy/memmove` calls), zIO unmaps the destination buffer, forgoes the copy, and tracks the destination buffer as *intermediate* (②). Some buffer locations may not be page aligned, in which case, *buffer fringes* have to be copied (`app_buf3` in Figure 3 is unaligned, causing copies in ③ and ④, where it is used as destination and source buffer, respectively). To provide consistency when applications access intermediate buffers, zIO leverages page faults. If a page fault to any intermediate buffer occurs, zIO finds the original buffer location to resolve the page fault with the appropriate data by lazily copying faulted pages (⑤). Finally, when tracked data is written to another IO stack (e.g., `send/write` calls), zIO intercepts the call and provides the original buffer instead of the application-provided intermediate buffer, but including any intermediate data updates (⑥). Before we detail each of these mechanisms, we describe zIO's tracking granularity and data structure.

Page granularity copy elimination. To be able to provide data consistency via page faults, zIO eliminates copies only at page granularity. However, buffers may reside at any address in virtual memory. To resolve this issue, zIO will only eliminate the part of a copy that lies within page boundaries of the provided buffer (i.e., unaligned buffer start addresses are rounded up to the page boundary, while unaligned buffer end addresses are rounded down)—the *core buffer*. The *left and right buffer fringe*—the beginning and end of an application buffer that is beyond the core buffer page boundaries, respectively—is always copied. While this approach involves small copies for unaligned buffers, we find that it often helps performance. The left and right buffer fringe often contain headers and footers that applications are more likely to access than the core.

Intermediate buffer tracking via skiplists. zIO records the locations of all application data buffers containing IO data. As buffer tracking has to incur minimal overhead and records are frequently mutated, we choose a skiplist for probabilistic fast search and insertion. Each entry in the skiplist keeps track of the original buffer address, a corresponding core intermediate buffer address, the length of the core intermediate buffer as a number of base pages, the size of the left intermediate buffer fringe in bytes, a timestamp of the last copy (cf. §3.1.7), and a free flag (cf. §3.1.4, not shown in Figure 3). The skiplist is sorted by intermediate buffer address. We evaluate the performance of buffer tracking via skiplists in §5.2.1.

3.1.1 Input buffer recording. When data is read from an IO stack via a function or system call, zIO intercepts these operations. We have implemented intercepts for all common POSIX network and file system calls. According to its policy (cf. §3.1.6), zIO records the application-provided destination buffer as the original buffer, along with an identity core intermediate buffer (①). This record filters IO buffers for copy tracking—zIO only eliminates copies for data originally read from an IO stack.

3.1.2 Copy tracking and elimination. zIO identifies copies within the application by interposing on the standard library memory copy calls `memcpy` and `memmove`[1]. These calls take a source and destination buffer address, as well as a size (in bytes) to copy. On each call, according to policy (cf. §3.1.6), instead of executing the copy, we record in the skiplist the core

[1]Variations of these calls use `memcpy` and `memmove` in our standard C library. For other C libraries, variations may need to be explicitly intercepted.

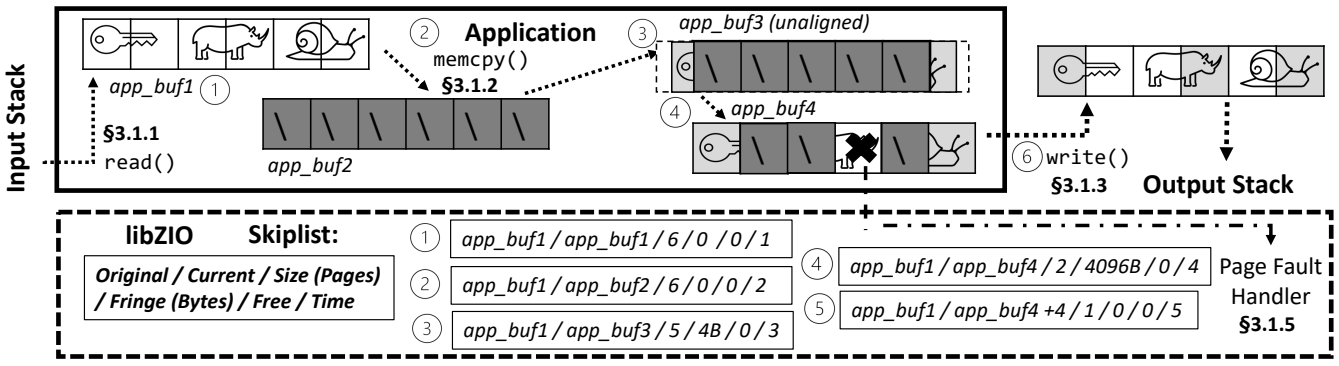


Figure 3. zIO application copy elimination example with an IO buffer spanning 6 pages. Original buffer pages are unshaded. Shaded pages are copied. Dark shaded pages are unmapped.

destination buffer and its size as intermediate location and size (e.g., ②, where `app_buf1` and `app_buf2` do not have a fringe).

To determine the original buffer location, we first use the core source buffer address to search through the skiplist to see if it falls within any existing intermediate buffers. If it does, we use that buffer's original buffer location, and, if this is the first time this original buffer is copied, zIO also remaps the core original buffer read-only to detect any application modification to it. If we find no intersecting intermediate buffer, then this data did not originate from IO (cf. §3.1.1) and we execute the copy, forgoing any tracking of this buffer. Finally, if the data originated from IO, we record the size of the left intermediate buffer fringe (③, where `app_buf3` is unaligned and has a left fringe of 4 bytes—it also has a right fringe, but we do not need to record it). The left buffer fringe is necessary to resolve access conflicts (cf. §3.1.5). If the destination location is within a buffer that is already tracked in the skiplist, the skiplist entry is updated with the new buffer information.

zIO unmaps the core intermediate buffer and registers it with userfaultfd to intercept application access. The union of left and right buffer fringes of original and intermediate buffer is copied. For example, if the source buffer has a left fringe of 4 bytes and the target buffer has no left fringe, then the left fringe of the target buffer becomes 4KB, as the original left fringe taints the first 4 bytes of what could have been a core page of the target buffer, making the entire page part of the fringe (④, where `app_buf3` has a left fringe of 4 bytes, `app_buf4` acquires a left fringe of 4KB).

The cost of unmapping intermediate buffers is often avoided or amortized. For example, buffers that are allocated on the heap are backed with physical memory and mapped only on first access (this *lazy memory allocation* is the default in Linux, for example). zIO can simply register these unmapped buffers with userfaultfd. Statically allocated buffers are often reused across requests instead of freed and reallocated. These buffers remain unmapped across requests if they are not otherwise accessed by the application. Upon reuse, zIO simply updates the skiplist when new IO data is processed.

3.1.3 Input to output buffer resolution. Whenever data is written to an IO stack, zIO interposes on the IO stack API and searches the skiplist to see if the core buffer being written intersects with any intermediate buffers tracked in the skiplist. If a match is found, zIO modifies the write operation to use any original buffer addresses recorded in the skiplist. This may result in a single IO stack source buffer location being transformed into multiple buffer locations (⑥, where shaded areas of the output are copied, unshaded areas are sourced from original buffer locations). If the IO stack API supports gather IO, we leverage that API to refer to the appropriate buffer pages when generating the output IO call. If the IO stack does not support gather IO, zIO breaks up the output call into multiple calls that refer to each individual buffer.

3.1.4 Freeing buffers. Finally, zIO interposes on `free`. This interposition allows zIO to look up and delete skiplist entries that are potentially no longer needed. If an intermediate buffer is freed that means we have successfully eliminated a copy; the contents of the buffer were not touched and the application has specified that it no longer needs it. At this point, the skiplist entry can be deleted and the memory region unregistered from userfaultfd. If an original buffer is being freed, the skiplist entry is only marked as freed to prevent use-after-free violations. Buffers marked as freed are deleted upon garbage collection (see below).

3.1.5 Access conflict resolution. When a core intermediate buffer is touched by the application, it will trigger a page fault. zIO looks up the faulting page number in the skiplist. zIO then maps the faulting page, potentially allocating it (lazy memory allocation), and copies the data from the original buffer, as recorded in the skiplist entry. zIO uses the left buffer fringe size to determine the byte offset of the intermediate core buffer, which is used as an offset into the recorded original buffer to determine the copy source address.

If a page at the beginning or end of a buffer is faulted in, it is removed from the tracked buffer core and thus copied going forward. If a page is faulted in the middle of a buffer, a

new skiplist entry must be created for the second section of the buffer, if it meets the appropriate size threshold (⑤). The original buffer is effectively split; the buffer before the faulting page is considered part of the originally tracked buffer and the buffer after the faulting page is a newly tracked buffer.

If a core original buffer is modified by the application, it also triggers a page fault. In this case, zIO walks the skiplist to determine any intermediate buffers derived from the core original buffer page. zIO copies the faulting original buffer page to the relevant intermediate buffers and resets the access permissions to the relevant original and intermediate pages.

3.1.6 Tracking policy. We determine experimentally (cf. §5.1.2) that for data buffers smaller than 16KB, the overhead of tracking outweighs any performance benefits from eliminating copies. Hence, we configure zIO to track and elide only sufficiently large copies (core buffer sizes of 16KB or larger).

There is also an overhead for handling page faults. We determine this experimentally (§5.1.4) under a number of conditions. For example, we find that if the ratio of bytes accessed by the application to bytes eliminated from copies exceeds 6%, we no longer see a performance benefit with a single application thread. This number can change with a different number of threads and is fully explored in §5.1.3. After these thresholds, we stop eliding copies for these buffers.

3.1.7 Intermediate buffer garbage collection. zIO avoids tracking an arbitrary number of entries to prevent memory exhaustion and skiplist performance reduction. For example, tracked intermediate buffers may be kept indefinitely in memory by the application, causing skiplist entries to accrue. Hence, skiplist entries are garbage collected periodically (once every second in our prototype). For each collected skiplist entry, we must fill any intermediate buffers with consistent data. This is done via the same process as conflict resolution. The region is mapped and the data is copied from its original location at the appropriate offset. Buffers marked free can be freed immediately.

zIO's garbage collection policy collects intermediate buffers that have been least recently used in copies. A timestamp on each skiplist entry (not shown in Figure 3) keeps track of the last time the entry was involved in a copy. If the skiplist grows beyond a threshold, zIO collects the least recently used entries.

3.2 IO Stack API Copy Elimination

Simply linking zIO when kernel-bypass IO stacks are used already provides transparent zero-copy IO. However, we can achieve further performance benefits by modifying these IO stacks to integrate with zIO more tightly. We now describe how we integrate zIO with kernel-bypass IO stacks to optimize IO stack API copy elimination.

Kernel-bypass IO stacks are a good fit for zIO, as they communicate with the application via shared library calls and shared memory—mechanisms that zIO can transparently process at user-level—rather than system calls. We choose the TAS [18] and Strata [20] kernel-bypass network and storage stacks, which are state-of-the-art. Strata, in particular, is a good fit, as it uses a per-process operation log in NVM, mapped into userspace, to persist file writes. zIO transparently intercepts Strata's memory copies into this log and can provide transparent copy elimination, provided that the original buffer already resides in NVM.

Input API copy elimination. POSIX file and socket input calls (e.g., `read` and `recv`) require applications to provide a buffer that input data is copied into. In TAS and Strata, these library calls internally call `memcpy` to copy from an IO stack internal buffer to the application-provided buffer. zIO transparently tracks and eliminates this copy across the IO stack API (cf. §3.1). As the source buffers are IO stack-private, we do not need to protect the original source data buffer by remapping it read-only. Instead, we modify the IO stacks to execute zIO's garbage collection protocol for any tracked buffers that the IO stack intends to free or overwrite. To prevent this from happening frequently, we can configure the IO stack internal buffers to be sufficiently large. For example, socket receive buffers can be resized to hold at least the expected size of input data per IO request.

Output API copy elimination. POSIX file and socket output calls (e.g., `write` and `send`) require applications to provide a source buffer that output data is copied from. As with the input API calls, zIO already transparently eliminates stack-internal memory copies. As output buffers are IO stack-private, no unmapping is necessary. Instead, we modify the IO stacks to fetch the original buffer locations from zIO when the output data is processed. For example, when TAS sends payload from the socket transmit buffer or when Strata "digests" [20] the update log. When zIO has to resolve copies due to mis-speculation or garbage collection, the relevant output buffer fields are simply filled in with the appropriate data. When the IO stacks ask zIO for original buffer locations, filled output buffers will not be marked as intermediate.

3.3 Optimistic Input Persistence

To realize optimistic input persistence for end-to-end IO copy elimination when data is persisted in NVM by a storage stack, we simply have to ensure that the original data already resides in NVM. zIO automatically detects the type of memory backing a virtual memory mapping. If original and intermediate buffers are backed by NVM, zIO can eliminate and track any copies among the buffers, while ensuring persistence. We describe here how we use this feature to realize *optimistic network receiver persistence*, where incoming data from the network does not need to be copied to storage.

Optimistic network receiver persistence. TAS uses shared memory for socket receive buffers between its TCP fast-path process and processes linking the kernel-bypass libTAS library. The fast-path writes incoming payload directly into

socket receive buffers residing in this shared memory. We can realize optimistic network receiver persistence simply by mapping the socket receive buffers into NVM. zIO will detect that original buffers are backed by NVM and eliminate copies end-to-end to the Strata update log, which also resides in process-local NVM.

3.4 Discussion

Huge pages. Huge pages (pages larger than the system's base page size) are desirable for improved memory address translation performance. However, tracking IO buffers requires fine-grained page protection, as tracked buffers may be smaller than the huge page size. In this case, zIO's fine-grained page mapping requests force the OS to break huge pages into base page mappings. Indeed, an investigation of the Redis YCSB benchmark with 512KB value size (cf. §5.2) shows that Linux with transparent huge page (THP) support maps 40% of Redis' working set with huge pages when zIO is not used, while mapping only 35% of the working set with huge pages when zIO is used.

Unfortunately, if the application stores IO buffers in reserved huge page memory using Linux's `hugetlbfs` mechanism, fine-grained page protection is disallowed and zIO can only track buffers at huge page granularity. Note that Linux could technically allow fine-grained protection for reserved huge page memory, while still allocating memory at huge page granularity. This would be compatible with zIO.

Luckily, transparent zero-copy and huge pages do not need to be at odds. zIO operates on the assumption that tracked IO buffers are seldom touched by applications. Hence, leveraging fine-grained page protection for tracking IO buffers does not impact application performance in the common case, as these mappings are seldom exercised. On mis-speculation, zIO's policy reverts to copying IO buffers and the OS may again map them with huge pages. This may happen transparently when THP support is enabled in the OS. Our application benchmarks run with THP, showing that transparent zero-copy IO still outperforms any potential slow-down from fine-grained page protection.

Linux kernel IO stack API copy elimination. While we present IO stack API copy elimination with kernel-bypass stacks (§3.2), we believe it is possible to provide IO stack API copy elimination for the Linux kernel IO stacks in certain cases by leveraging Linux's zero-copy IO APIs (cf. §2.4). For example, using Linux's zero-copy socket receive API (cf. §2.4), zIO can memory map kernel TCP socket receive buffers into user-private memory when sockets are created. It can then intercept application `recv` calls and track the target application buffer as an intermediate buffer, with the private socket buffer mapping as the original. This eliminates the IO stack API copy for `recv`, similar to our integration with TAS, as described in §3.2. Network receiver persistence may also be realizable, albeit with kernel modifications, by mapping socket buffers into a file stored in NVM and then using the `FICLONERANGE` ioctl to remap core data buffers to their final destination upon input to output resolution to a file. We leave IO stack API copy elimination for the Linux kernel IO stacks for future work.

4 Implementation

Our zIO implementation consists of two key components. The first component is tracking data through an application and eliminating copies along the way. The second component is closely integrating this tracking with the kernel-bypass network and storage stacks TAS and Strata, respectively, to provide transparent zero-copy across IO stack APIs, as well as optimistic input persistence.

Application copy elimination. This component of zIO is written in 1,608 lines of C code and is dynamically loaded with `LD_PRELOAD`.

IO stack API copy elimination. To integrate zIO with TAS and Strata to provide IO stack API copy elimination, we modify 184 lines of code in TAS and 66 lines of code in Strata.

5 Evaluation

We analyze zIO's performance via a number of experiments based on a multi-threaded IO microbenchmark, using network and storage stacks, and varying relevant IO and copy parameters. We also evaluate zIO with the IO-intensive applications Redis [21], MongoDB [25], and Icecast [37]. We compare zIO to Linux and kernel-bypass IO stacks without any copy optimizations.

Our evaluation answers the following questions:

- What is the impact of copies on IO performance? What benefits to IO processing throughput does zIO provide by transparently eliminating copies? How do the number of copies per IO (§5.1.1), IO size (§5.1.2), and number of IO threads (§5.1.3) affect the observed performance?
- What are the overheads zIO introduces by tracking data? How do overheads increase as applications touch the data they copy, causing zIO to mis-speculate? How effective is zIO's tracking policy in avoiding mis-speculation? (§5.1.4)
- How do zIO performance improvements break down into its mechanisms? By how much can we improve IO performance when employing optimistic input persistence with NVM? (§5.2.1)
- What benefits to IO processing throughput and latency does zIO provide by eliminating copies within IO-intensive applications, such as Redis (§5.2), Icecast (§5.3), and MongoDB (§5.4)? In what situations might zIO hurt application performance (§5.3.1)?
- How does zIO perform compared to zero-copy IO APIs, such as memory mapped files and `sendfile`? (§5.3.1)

Evaluation platform. We run our evaluation on a single socket of a dual-socket Intel Cascade Lake-SP system running at 2.2GHz with 24 cores per socket and a 100 GbE ConnectX-5

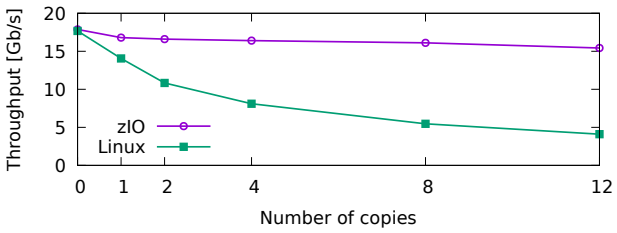


Figure 4. Linux throughput versus zIO application IO copy elimination (512KB IO size).

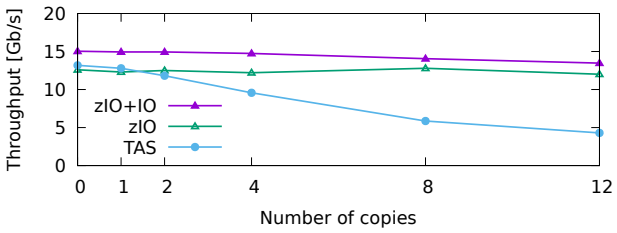


Figure 5. TAS throughput versus zIO application and IO stack API copy elimination (512KB IO size).

NIC. Each socket has 192 GB of DDR4 DRAM and 3 TB of Intel Optane DC NVM. To leverage all 6 memory channels per processor, there are 6 DIMMs of DRAM and NVM per socket. The machine runs Fedora 27 with Linux kernel version 5.10.0. We use the latest master branches of TAS [3] and Strata [4].

5.1 Microbenchmarks

We quantify the overhead introduced by copies of IO data and the benefit that zIO provides for various IO parameters via a simple echo server benchmark. Our evaluation setup is the same as in §2.2, but in place of Redis we run a simple TCP echo server that echoes client messages back to the sender. To simulate IO-intensive application processing, our echo server can make a configurable number of copies to the IO data. Beyond the number of copies, we also vary other IO parameters, such as IO size, fraction of IO data accessed, and number of echo server threads. We report the average echo server throughput, measured at the client, over 3 runs for each configuration, using the steady-state throughput of each run.

5.1.1 Number of Copies. We first evaluate IO performance with a varying number of copies of the IO data made before it is echoed. We compare four scenarios: Vanilla Linux (Linux), Linux with zIO application copy elimination (zIO), vanilla kernel-bypass (TAS), kernel-bypass with zIO application copy elimination (zIO), and kernel-bypass with zIO application and IO stack API copy elimination (zIO+IO). We run this experiment with 512KB IO, using a single server thread. For each run, we vary the number of times the request is copied before being echoed.

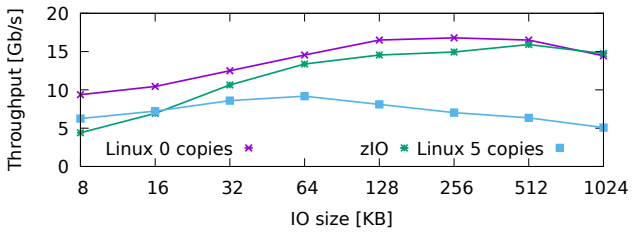


Figure 6. zIO throughput versus Linux with 0 and 5 copies.

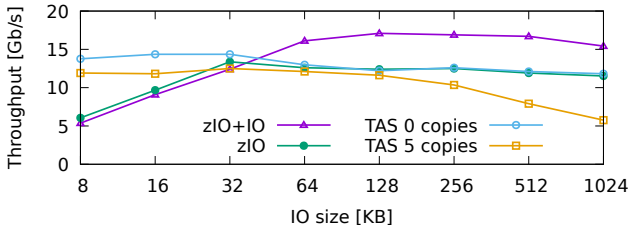


Figure 7. zIO throughput versus TAS with 0 and 5 copies.

Figures 4 and 5 present the results. We can see that an increasing number of application IO copies decreases the achieved throughput for Linux and TAS networking[2], due to the involved copying overhead. Kernel-bypass maintains high throughput with more copies than Linux, as more CPU cycles are available for copies due to the lighter-weight kernel-bypass network stack. zIO maintains performance close to the configuration without copies for both stacks, showing that it successfully eliminates these copies with negligible overhead. With 12 copies, zIO improves throughput by 3.8× with Linux and by 2.8× with TAS. Finally, zIO+IO improves throughput by up to 21% versus zIO by additionally eliminating IO stack API copies.

5.1.2 IO Size. We next investigate how IO size affects performance, using a single echo server thread. To evaluate the overhead of tracking small IO, we disable zIO's IO size threshold for this benchmark, causing zIO to always track buffers and eliminate copies. We vary the IO size from 8KB to 1MB and evaluate two extreme copy scenarios (cf. Table 1): 5 application copies and 0 application copies. Figures 6 and 7 present the results.

zIO benefits large IO. Firstly, we can see that Linux has poor performance with small IO, but performance improves as IO size increases. TAS performs better with smaller IO size. This is expected, as kernel-bypass stacks are light-weight. When copies are involved, both Linux and TAS perform worse, in particular as IO size increases. This is also expected, as

[2]We consistently observe TAS throughput to be lower than Linux with large IO sizes. TAS is optimized for small IO and does not do the necessary batching to handle large IO efficiently.

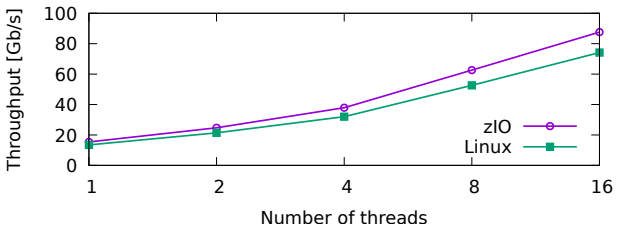


Figure 8. zIO scalability.

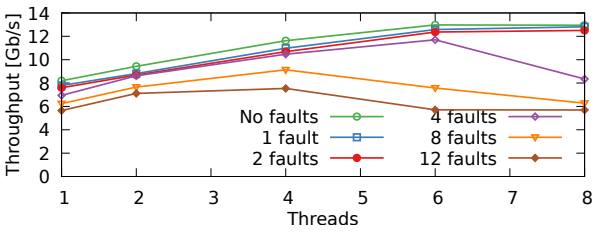


Figure 9. zIO scalability with page faults.

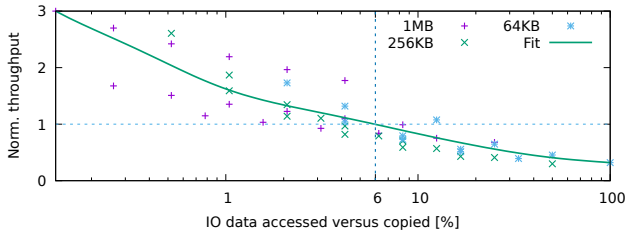


Figure 10. zIO throughput improvement under data access.

larger copies require more CPU time. zIO improves throughput by up to 2.9× with Linux and up to 2× with TAS as IO size increases, reaching zero-copy performance with IO sizes larger than 512KB for Linux and 32KB for TAS. zIO+IO improves throughput further, by up to 40% versus zIO, for a combined improvement of up to 2.7× versus TAS.

Limits of zIO with small IO. zIO transparent copy elimination is no panacea, as the overhead of zIO tracking with small IO limits throughput. For IO smaller than 16KB, zIO reduces throughput by up to 30% versus Linux. For IO smaller than 32KB, zIO reduces throughput by up to 49% versus TAS. IO sizes smaller than 8KB would incur even further throughput reduction. Based on this measurement, we set zIO's tracking policy to avoid tracking IO buffers smaller than 16KB (cf. 3.1.6).

5.1.3 Scalability. We evaluate two scalability aspects. zIO tracking scalability and the impact of page faults.

zIO tracking. We configure the echo server to make 1 application copy of each 512KB IO buffer and vary the number of server threads. Each thread handles a private pool of clients and uses private IO buffers. Figure 8 shows that zIO improves throughput scalability over Linux by up to 19% due to copy elimination. Copies pollute the CPU caches, causing Linux's scalability to be impacted.

Page faults. Page faults can affect scalability when faulting pages are mapped, requiring TLB shootdowns. In theory, information about newly mapped pages may be lazily synchronized among TLBs, avoiding TLB shootdowns. Other cores accessing the same unmapped page simply fault on the stale TLB information, synchronizing the TLB at this moment. Most IO-intensive applications use thread-private IO buffers and access across cores is rare. Unfortunately, Linux does not support lazy mapping of pages. Hence, page faults do affect scalability.

To show this effect, we configure the echo server to access a number of pages of each 512KB IO buffer, without application copies, and vary the number of server threads. We supply the same IO buffer each time, requiring zIO to unmap it for each IO request. The results can be found in Figure 9. We can see that, up to 2 page faults, server throughput still scales well. Increasing the number of page faults per IO beyond this point starts limiting server throughput due to TLB shootdowns. With Linux modifications, many of these TLB shootdowns could be avoided.

5.1.4 Mis-speculation. To evaluate the impact of zIO mis-speculation on performance, we configure the echo server to access a number of bytes in each IO request before echoing a response. We run this experiment under a variety of IO sizes (64KB, 256KB, and 1MB) and copies (1, 3, and 6), using the Linux network stack.

Figure 10 presents the results as a scatter plot, where we compare zIO throughput improvement over vanilla Linux to the ratio of IO bytes accessed versus elided in copies. This ratio clearly limits zIO's throughput improvements. Applications accessing copied IO data means that zIO mis-speculated. zIO has to resolve the elided copies for the accessed data, which incurs a performance penalty. Less IO data accessed implies better performance improvements. At the same time, more IO data elided in copies also implies better performance improvements and creates headroom for mis-speculation. Fitting a Bezier curve to the scatter plot shows that zIO improves throughput when the ratio of data bytes accessed by an application versus data bytes elided in copies is less than 6%. Above 6%, overheads created by zIO mis-speculation decrease throughput. As an example, for an input buffer of size 200KB that is copied twice, the application may incur up to 6 page faults before output to still yield a speed-up. If the same buffer is copied 4 times, up to 12 page faults are permissible.

5.2 Redis

We evaluate how zIO improves Redis throughput with Linux and kernel-bypass IO stacks (TAS and Strata). Our benchmark setup is identical to the one presented in §2.2. We evaluate two

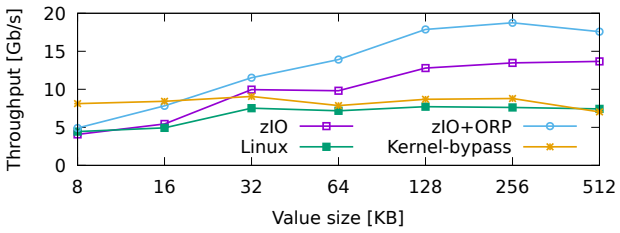


Figure 11. Redis throughput (100% SET).

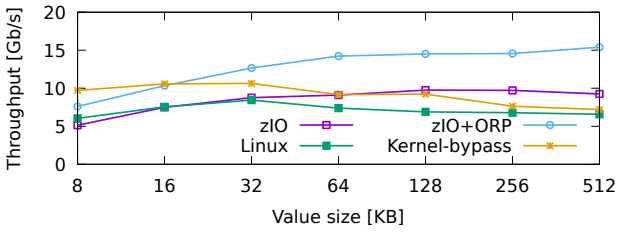


Figure 12. Redis throughput YCSB A (50% SET, 50% GET).

benchmark configurations: 1) 100% SET, and 2) YCSB Workload A, which has a distribution of 50% SETs and 50% GETs. We vary the value size over independent runs for each of these configurations. In addition to zIO's improvement over vanilla Linux with application copy elimination, we investigate the performance of zIO with additional optimistic receiver persistence and IO stack API copy elimination (zIO+ORP) over kernel-bypass IO stacks. The zIO size threshold is disabled for these experiments; enabling it would allow zIO to match vanilla IO stack performance for smaller values, evaluated in §5.2.1.

We first look at 100% SET throughput. This case involves 2 IO copies (one from the network and one to storage), as well as 4 IO application copies per request (cf. Table 1). The results can be found in Figure 11. zIO with Linux eliminates all application copies, which allows for a throughput improvement of up to 1.8×, especially for larger values. zIO+ORP with kernel-bypass IO stacks improves performance by up to 2.5×, as the IO paths consume noticeably less CPU time.

We now look at YCSB workload A, with 50% GET requests and 50% SET requests. These results can be found in Figure 12. As the 50% GET requests require fewer application copies, zIO with Linux provides less of a performance improvement than in the first benchmark, up to 1.3×. However, GET requests provide an opportunity for zIO+ORP to eliminate IO stack API copies, maintaining a speedup of up to 2× over vanilla kernel-bypass.

5.2.1 zIO Performance Breakdown. We use the Redis 100% SET benchmark to break down the performance contributions of zIO. To do so, we evaluate zIO throughput with kernel-bypass IO in two IO size configurations, progressively

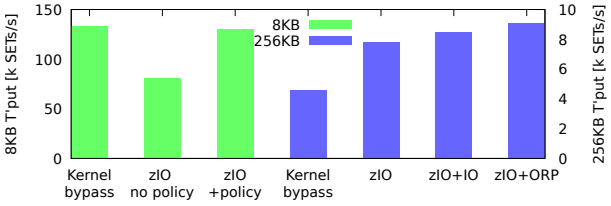


Figure 13. zIO performance breakdown.

	Storage to net **Throughput**	Net to net **Listeners**
Kernel-bypass	0.89 GB/s (1.00×)	812 (1.00×)
zIO+IO	1.08 GB/s (1.25×)	944 (1.16×)

Table 3. Icecast throughput.

enabling different zIO optimizations. These results can be found in Figure 13.

The first configuration uses 8KB SET requests. We evaluate zIO with and without its tracking policy, which applies a 16KB size threshold (§3.1.6). We can see a drastic slowdown of 40% when zIO does not apply this policy, due to the overhead of tracking small IO. Enabling zIO's policy instead copies the IO buffers and attains a negligible slowdown of 2% versus vanilla kernel-bypass.

We further evaluate a configuration with 256KB SET requests. When eliminating application copies, zIO provides a speedup of 1.7×. When adding IO stack API copy elimination, zIO+IO improves performance by another 9%. Adding optimistic receiver persistence in zIO+ORP finally improves performance by another 7%, for a combined improvement over vanilla kernel-bypass of 2×.

Intermediate buffer tracking overhead. We investigate the overhead of buffer tracking via zIO's skiplist. For the same 100% SET request Redis configuration, we find an average of 5 skiplist entries per client connection. With 64 clients, we measured a maximum of 640 entries in the skiplist over the duration of the benchmark. For this scenario, we measure the average skiplist operation latency for lookup and insert to be 190ns. This confirms that intermediate buffer tracking via skiplists is lightweight.

5.3 Icecast

Icecast [37] is an audio broadcasting service. Icecast can stream audio from a source client to a number of listener clients or read data from a local file and serve it to a number of listener clients via HTTP. Table 1 shows that Icecast makes no application copies, but it uses the IO stack APIs. We evaluate both Icecast configurations, providing insight into network to network and storage to network performance. We use the kernel-bypass IO stacks for our evaluation, as they support IO stack API copy elimination. The results are shown in Table 3.

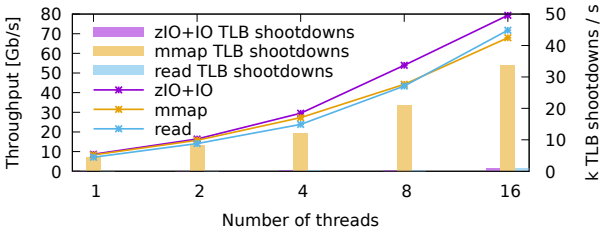


Figure 14. Icecast throughput scalability.

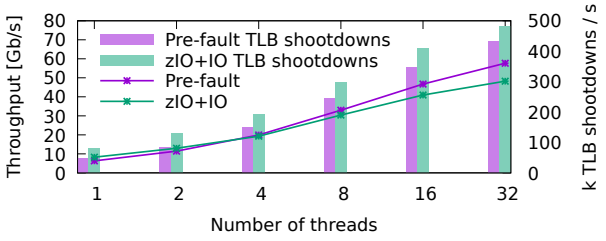


Figure 15. Icecast scalability with pre-faulted buffers.

Storage to network. We configure Icecast to broadcast a 1.1MB audio file to a number of listener clients. We evaluate the amount of audio that a single Icecast file-serving thread can deliver. Icecast reads and sends the audio file in a configurable chunk size, which we set to 64KB. Listener clients request the audio stream via `curl-loader` [16], a HTTP benchmark tool. We connect enough clients to saturate Icecast server throughput and measure throughput over a 30 second period. We can see that zIO+IO improves Icecast maximum throughput by 1.25× by eliminating IO stack API copies, freeing CPU cycles for audio streaming.

Network to network. Next, we evaluate Icecast receiving data from a single client and broadcasting it to a number listeners via a single relay thread. We configure Icecast to relay 64KB at a time and measure the number of concurrent listeners that Icecast can broadcast to. We see a zIO+IO improvement of 1.16× by eliminating IO stack copies. Icecast uses a static buffer for relay, which remains unmapped across IO chunks. This allows zIO+IO to eliminate IO stack copies with minimal overhead.

5.3.1 Scalability. Icecast is a single-threaded application when serving local files to listeners. To evaluate IO-intensive application scalability with zIO, we modify Icecast to create a thread-pool, where each thread can handle listener client HTTP requests from storage via a thread-local IO buffer. This configuration makes Icecast behave like a web server, such as Apache [6]. This version of Icecast is using the `read` system call to read from each file (read).

zIO scalability versus zero-copy IO interfaces. Web servers (like Apache) often use zero-copy IO interfaces to accelerate service, such as memory mapped files and the `sendfile` system call. To compare application performance with a zero-copy IO interface to that of zIO's transparent zero-copy IO, we create a version of Icecast that maps a requested file into memory (cf. §2.4) and sends data to the clients from the memory-mapped file via the socket `send` call (mmap). Memory mapping each requested file eliminates an IO stack copy on input, but also incurs a TLB shootdown. Common usage (cf. Apache) of the mmap API unmaps each file after serving it, incurring another TLB shootdown. zIO+IO can eliminate copies without having to incur TLB shootdowns if buffers are re-used and remain untouched in the common case.

We evaluate these configurations with a 512KB audio file with an increasing number of threads and measure throughput, as well as the number of TLB shootdowns. These results are found in Figure 14. We can see that zIO+IO consistently performs the best, as it does not incur TLB shootdowns. For a small number of threads, memory mapping input files performs similarly to zIO+IO. However, as the number of threads increases, the number and cost of performing TLB shootdowns increases, which negatively affects mmap performance. The number of TLB shootdowns when using `read` and zIO+IO are negligible, as no memory mapping calls happen in the common case. zIO outperforms memory mapping of input files by up to 17%.

Finally, we evaluate versions of Icecast using the Linux `sendfile` API to transmit files to listeners. The first version uses mmap to memory map each file to validate its header before using `sendfile` to transmit it. The second version uses the `read` system call to read the file's header. These versions cannot use the kernel-bypass IO stacks, as `sendfile` is kernel-specific, and `read+sendfile` performs up to 7% worse than zIO+IO, while `mmap+sendfile` performs up to 30% worse than zIO+IO. The scalability trend of `read+sendfile` follows that of zIO+IO, while `mmap+sendfile` scales similarly to mmap.

zIO scalability with pre-faulted buffers. We have already evaluated zIO scalability when buffers are touched, incurring page faults (§5.1.3). zIO can detect these cases and stop copy elision (§3.1.6). However, if the application causes page faults before buffers are tracked by zIO, for example by pre-faulting mapped memory (cf. `MAP_POPULATE` flag for `mmap`) before using it to buffer IO, then zIO can incur TLB shootdowns by unmapping these buffers for tracking.

To evaluate this scenario, we modify Icecast to pre-fault the IO buffer before reading into it via `read` and unmapping it after it was sent over the network (pre-fault). This forces zIO+IO to unmap the IO buffer to track potential access. We run these two configurations with a 512KB audio file, a 32KB chunk size, and an increasing number of threads. We measure throughput and TLB shootdowns for both cases. We present these results in Figure 15. With a small number of threads, zIO+IO outperforms pre-fault, as it still eliminates copies in the IO stack API. However, as the number of threads increases, performance is affected by the additional TLB shootdown overhead and

zIO+IO performance degrades. Note that pre-faulting memory causes TLB shootdowns by itself and the scalability of this scenario is already limited.

5.4 MongoDB

We run MongoDB [25] on Linux, with and without zIO. We connect a client over the network running the YCSB [10] load phase and measure request throughput with 1MB values, divided into 10 fields. The YCSB load phase is a workload with 100% inserts of a uniform random distribution. We repeat this benchmark 5 times and report the average throughput for each configuration.

We find that zIO is not able to provide a performance benefit for this workload, with a performance of 191 requests/s compared to 194 for Linux without zIO. zIO is disabling all optimizations due to a large number of page faults. We find that the page faults are generated by MongoDB reading each inserted value in its entirety to calculate a checksum before writing it to the file system.

If we modify MongoDB to skip checksum calculation, zIO is able to eliminate 2 out of 3 application copies (cf. Table 1). Similar to Redis (cf. Table 2), MongoDB copies the inserted value first into an in-memory B-tree (similar to Redis' copy A_2) and then into a log (copy A_3). Finally, MongoDB reallocates the IO buffer, causing a copy, before inserting it into an on-disk index. All three copies are initially elided by zIO, the file system writes complete and their buffers are freed. However, the next IO request re-uses the original IO buffer, forcing zIO to execute the elided copy of the previous buffer to the B-tree data structure. zIO achieves a throughput of 222 requests/s, a 6% improvement over Linux' throughput of 209 requests/s.

We also run MongoDB with the TAS kernel-bypass network stack, allowing us to use zIO+IO to elide an IO stack API copy in `recvmsg` that MongoDB uses to read data from the network. Doing so additionally implies that original buffer reuse, which is now internal to the IO stack and directly communicated to zIO+IO, is lighter weight, as it is not initiated via a page fault. TAS without zIO+IO achieves a throughput of 191 requests/s, while TAS with zIO+IO achieves a throughput of 229 requests/s, a 19% performance improvement.

6 Related Work

In this section, we cover related work beyond the zero-copy IO APIs studied in §2.4.

Zero-copy networked storage. Reflex [19] is a networked storage system designed to provide fast access to remote flash devices. Reflex gains performance by eliminating software copies between network interface cards and flash storage. Unlike zIO, ReFlex does not focus on eliminating application-level or IO stack API copies.

Hardware-accelerated serialization. Recent work has looked at accelerating serialization with help from hardware. Zerializer [35] proposes DMA hardware with data transformation logic to offload serialization. Breakfast of Champions [29] proposes using existing scatter-gather capabilities of NICs to offload serialization. Unlike these works, zIO provides zero-copy without assuming specialized hardware and can eliminate application copies beyond those needed for serialization.

Custom user-level IO stacks. Sandstorm [23] addresses the idea of specially tailoring user-level IO stacks to meet the specific needs of applications to maximize performance, including zero-copy. However, similar to cross-stack APIs, these customizations are not transparent. Either the IO stack has to be modified to work with the application, the application has to be modified to use new APIs, or both. zIO offers transparent cross-stack zero-copy.

7 Conclusion

We present zIO, a transparent zero-copy IO mechanism for unmodified IO-intensive applications. zIO tracks IO data through the application, eliminating copies that are unnecessary while maintaining data consistency. We implement zIO as a user-space library, supporting Linux kernel and kernel-bypass IO stacks. We evaluate zIO with IO-intensive applications, like Redis, Icecast, and MongoDB. zIO improves application throughput by up to 1.8× with Linux, as well as by up to 2.5× with kernel-bypass IO stacks with optimistic network receiver persistence.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Dan Tsafrir, for their helpful comments and feedback. This work was supported by NSF grants 2226057, 2227066, and 2227132.

References

[1] sendfile(2)—linux manual page. https://man7.org/linux/man-pages/man2/sendfile.2.html.

[2] Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, September 2014.

[3] https://github.com/tcp-acceleration-service/tas, 2020. Commit d3926baf6ad65211dc724206a8420715eb5ab645.

[4] https://github.com/ut-osa/strata, 2020. Commit f368da4cefe874e1b31a19df7c6436b48f489381.

[5] userfaultfd(2). http://man7.org/linux/man-pages/man2/userfaultfd.2.html, February 2020.

[6] Apache. Apache HTTP Server, 2022. https://httpd.apache.org/.

[7] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 65–77, 2021.

[8] J.S. Chase, A.J. Gallatin, and K.G. Yocum. End system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[9] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *USENIX Annual Technical Conference*, January 1996.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[11] Jonathan Corbet. Zero-copy networking, 2017. https://lwn.net/Articles/726917/.

[12] Jonathan Corbet. Zero-copy TCP receive, 2018. https://lwn.net/Articles/752188/.

[13] Google. Protocol buffers, 2008. https://developers.google.com/protocol-buffers.

[14] Google. gRPC, 2016. https://grpc.io.

[15] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A network programming interface for non-volatile main memory. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, pages 17–33, 2018.

[16] Robert Iakobashvili and Michael Moser. curl-loader, 2007. http://curl-loader.sourceforge.net/index.html.

[17] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation*, pages 1–16, 2019.

[18] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the 14th EuroSys Conference*, pages 1–16, 2019.

[19] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. *SIGARCH Comput. Archit. News*, 45(1):345–359, April 2017.

[20] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[21] Redis Labs. Redis, 2022. https://redis.io/.

[22] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 90–103, 2019.

[23] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. *SIGCOMM Comput. Commun. Rev.*, 44(4):175–186, August 2014.

[24] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 249–262, 2016.

[25] MongoDB. MongoDB, 2022. https://www.mongodb.com/.

[26] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. TensorFlow-Serving: Flexible, high-performance ML serving. In *Workshop on ML Systems at NIPS*, 2017.

[27] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, February 2000.

[28] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 33(4):1–30, 2015.

[29] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of champions: Towards zero-copy serialization with NIC scatter-gather. In *Proceedings of the 23rd USENIX Conference on Hot Topics in Operating Systems*, pages 199–205, 2021.

[30] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.

[31] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[32] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: a virtual memory management technique for zero-copy communication. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 308–314, 1998.

[33] Vesoft, Inc. Nebula graph, 2019. https://nebula-graph.io/.

[34] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 307–320, 2006.

[35] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the 23rd USENIX Conference on Hot Topics in Operating Systems*, pages 206–212, 2021.

[36] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment*, 12(6):624–638, 2019.

[37] xiph. Icecast, 2022. https://icecast.org/.

[38] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.

Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning

Tej Chajed
MIT CSAIL

Joseph Tassarotti
Boston College

Mark Theng
MIT CSAIL

M. Frans Kaashoek
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

Abstract

This paper develops a new approach to verifying a performant file system that *isolates crash safety and concurrency reasoning* to a transaction system that gives atomic access to the disk, so that the rest of the file system can be verified with *sequential reasoning*.

We demonstrate this approach in DaisyNFS, a Network File System (NFS) server written in Go that runs on top of a disk. DaisyNFS uses GoTxn, a new verified, concurrent transaction system that extends GoJournal [9] with two-phase locking and an allocator. The transaction system's specification formalizes under what conditions transactions can be verified with only sequential reasoning, and comes with a mechanized proof in Coq [37] that connects the specification to the implementation.

As evidence that proofs enjoy sequential reasoning, DaisyNFS uses Dafny [26], a sequential verification language, to implement and verify all the NFS operations on top of GoTxn. The sequential proofs helped achieve a number of good properties in DaisyNFS: easy incremental development (for example, adding support for large files), a relatively short proof (only 2× as many lines of proof as code), and a performant implementation (at least 60% the throughput of the Linux NFS server exporting ext4 across a variety of benchmarks).

1 Introduction

File systems are important to implement correctly because applications rely on them to safely store user data. Formal verification offers a promise of showing that the implementation of a file system always meets its specification, including a crash safety property that says the file system recovers correctly from a sudden crash and reboot. However, efficient implementations are internally complicated, especially because they support concurrency and aim to minimize disk writes. Complexity makes the code more error-prone and motivates the desire for formal verification, but also poses a challenge: how can a proof cover concurrency, crash safety, and functional behavior while remaining tractable for a program the size of a file system?

The main contribution of this paper is a new approach to verifying a file system that *isolates crash safety and concurrency reasoning* to a transaction-system implementation. This use of a transaction system wraps the file-system data structures and logic inside a transaction, and permits *sequential reasoning* for the body of each transaction. Sequential reasoning keeps the proof burden manageable even with an efficient implementation that supports many features, such as large files and in-place updates of serialized metadata.

There are three challenges in realizing this approach. The most important lies at the interface between the transaction system and the file system: intuitively, transactions make things simpler, but how do we exploit this for a proof engineer verifying the code running in a transaction? This paper proves a *simulation transfer theorem* that formalizes how the proof engineer can verify the body of each transaction using sequential reasoning, and yet still obtain a proof about concurrent and crash behavior, due to the use of a verified transaction system. This specification and its proof are not specific to the file system written on top and could be applied to another storage system implemented using transactions. We use the transaction system with file-system code verified using Dafny [26], a verification-oriented programming language that is limited to sequential reasoning but in exchange has good automation.

The second challenge is how to implement and verify the transaction system itself. The performance and concurrency of the overall system can only be as good as the transaction system, so efficiency and fine-grained locking are important. To that end we implement a new transaction system called GoTxn by extending GoJournal [9] (a verified journaling system) with two-phase locking. GoJournal's specification guarantees crash safety but requires the caller to implement concurrency control (enforced with separation logic) to achieve atomicity. In proving GoTxn we give a new separation-logic proof of two-phase locking's correctness based on local reasoning rather than the typical textbook approach that reasons about the global conflict graph for a set of transactions. GoTxn cannot make arbitrary transactions appear atomic (for example, if they access global variables), and so the specification for GoTxn applies only to a carefully formalized subset of "safe" transactions that access shared state only through the transaction system.

The third and final challenge is how to implement the file system using only transactions. GoTxn's safety restriction would appear to preclude an in-memory allocator since it re-

quires other shared state, which we address by incorporating allocation with a non-deterministic specification into GoTxn, which is then used in the file system by validating the allocator's output. For sequential reasoning each operation must be implemented as a single transaction, but operations like removing a file can require a large number of disk writes that might not fit in a transaction. We implement freeing using multiple transactions; a first transaction logically deletes a file, and then asynchronously the implementation can run transactions that recover space from the file but have no other visible effect.

The verified artifact from this work is DaisyNFS, which implements a Network File System (NFS) server in Go on top of a bare disk and comes with a proof that clients observe that each operation follows the NFS specification as laid out in RFC 1813 [4]. Operations appear atomic despite concurrency and crashes. Clients can use the Linux or macOS NFS clients to mount DaisyNFS like any other file system and interact with it using the usual POSIX API. As an end-to-end check that our formalization of NFS is accurate and the implementation is reasonably complete, we tested with both Linux and macOS clients running a variety of programs.

A benefit of this file-system design is that it permits using the sharpest tool for each part of the proof: while we use Perennial [9], a program logic for crash safety and concurrency embedded in Coq, for the transaction system's proof, we use Dafny [26], a verification-aware programming language with powerful automation, for the file-system operations. Dafny is a purely sequential language, but we are able to use it despite this limitation due to the transaction system's proof. The value of sequential proofs can be seen in the proof-to-code ratio for the transaction system, which is 20×, versus the Dafny proofs which required about 2× as many lines of proof as code. Further evidence can be seen in the incremental development of DaisyNFS, which we elaborate on in §9.4.

To evaluate DaisyNFS's performance, we compare it to that of the Linux NFS server exporting an ext4 file system. DaisyNFS achieves within 90% of the throughput of Linux with the ext4 `data=journal` option (which gives the same crash-safety guarantees as DaisyNFS) across a variety of benchmarks both on an NVMe and in-memory disk, and at least 60% on the most challenging ones. The comparable performance is due to the efficiency of GoJournal and adding little overhead in the file-system code (e.g., updating data structures in place to avoid copying). We do note that ext4's default `data=ordered` mode can get about 60% better throughput for data-heavy workloads, at the cost of weaker guarantees on crash.

The contributions of this paper are:

- Formalization of a *simulation-transfer theorem* that captures how the transaction system provides sequential reasoning (§5.1) for any system implemented using a transaction per operation.
- A proof that the simulation-transfer theorem holds for the GoTxn implementation (§6). This proof verifies two-phase locking using a new strategy based on *local reasoning* to connect to the GoJournal specification. For the theorem to be true, it needs a precisely formulated definition of *safe* transactions that access shared state through GoTxn in order to behave atomically.
- Techniques to implement a file system using GoTxn, including a validation approach to integrating in-memory allocation into GoTxn and an approach for splitting file removal into multiple transactions of bounded size.
- DaisyNFS, a concurrent, crash-safe file system that is verified in Dafny with sequential reasoning thanks to the above techniques. The Dafny proofs for the file-system code enjoy low overhead compared to the concurrent proofs for GoTxn (2× vs. 20×). A performance evaluation shows that DaisyNFS gets throughput at least 60% that of Linux ext4 exported over NFS for the most challenging benchmarks, and within 90% for many workloads.

Our approach and DaisyNFS have some limitations. The proof approach relies on transactions appearing to run sequentially, which prevents modifying state outside the transaction system. There are cases where that would get better performance in exchange for a more difficult proof. The transaction system does not have a proof of liveness, and we do not prove that transactions avoid deadlock. DaisyNFS does not support NFS unstable writes, which improve performance by not committing writes to stable storage until explicitly requested. Our NFS implementation does not cover some features, such as symbolic links, hard links, and paginated `READDIR`; we believe these features could be implemented and specified with the same approach but have not done so in our prototype.

This paper describes work that is part of the first author's Ph.D. thesis [5], which provides more detail. The thesis also describes the Perennial logic for verifying concurrent and crash-safe systems, the specification and proof of GoTxn (including GoJournal), and Goose, the tool we use to verify GoTxn's implementation written in Go. It goes into more detail about the DaisyNFS proof and evaluation as well.

2 Related work

Our main contribution is a way to use transactions to enable sequential reasoning for a concurrent file system. Our approach allows using Dafny and produces a file system that gets good performance. Prior work has also explored how to compose proofs across layers for modularity, to contain concurrency, or to cross between proof systems in complementary and distinct ways; none use transactions or any similar mechanism to isolate concurrency or crash safety reasoning.

2.1 Verifying storage systems

Directly related systems DaisyNFS directly builds upon GoJournal [9] to implement the transaction system, together with

its new version of the Perennial framework [8] that is used to verify the transaction system's proof. This infrastructure is a program logic designed for storage systems that need a combination of concurrency and reasoning about crashes at any time, built on top of the Iris framework [23] in Coq.

The transaction system differs from GoJournal in that the GoJournal specification requires the caller to prove that concurrent transactions do not attempt to read or write the same objects, whereas the transaction system guarantees this automatically with per-object locks. The specification styles are also different: whereas the GoJournal proof is a set of specifications within the Perennial logic, the transaction system's proof uses a more general refinement-based definition that we can apply to the Dafny code. This is necessary to combine the tools, since Dafny cannot express the GoJournal specification's concurrency restrictions directly.

Directly related applications In prior work with the Perennial framework, we verified a crash-safe, concurrent mail server under the assumption that the file system is crash-safe [8]. DaisyNFS is a crash-safe file system and its complexity is significantly larger than a mail server: the mail server is about 150 lines with a monolithic proof while DaisyNFS combines a transaction system (itself 1,600 lines) with a 4,000-line file system, each of which involve many intermediate abstractions.

The authors of GoJournal verify a simple NFS file server on top of GoJournal, but that server is not complete enough to run real applications (it supports only one directory and 4KB files). Furthermore, the simple NFS server does its own locking and so the proof must reason about concurrency, increasing the proof overhead compared to DaisyNFS.

Other verified file systems Flashix [33] is a verified file system for flash storage, recently extended to support concurrency by Bodenmüller et al. [2]. File-system operations are proven to be atomic using a variant of Lipton's movers [28] technique with additional conditions to ensure crash-atomicity [31]. In contrast, DaisyNFS proves once and for all that operations encapsulated in a transaction are atomic. Flashix uses per-file locks to enable concurrent file accesses, but the directory tree is protected by a single reader-writer lock, so operations creating or moving files cannot proceed concurrently. DaisyNFS's two-phase locking system allows operations to proceed in parallel if they access disjoint parts of the file system.

VeriBetrKV [18] is a verified key-value store similar to the one that underpins the BetrFS [22] file system. It uses Dafny for crash-safety reasoning but does not layer any file-system proof on top. This file-system design does not involve general transactions, so the code on top of the key-value store must still carry out crash reasoning. The system has I/O concurrency but no CPU concurrency.

AtomFS [39] is a verified concurrent file system that does not persist data. It uses a custom concurrent relational logic implemented in Coq. Because the system does not persist data, AtomFS does not have any transaction system and implements the file-system operations together with appropriate locking for concurrency control.

2.2 Concurrency verification

A number of verification frameworks address concurrency, including CIVL [20], CSPEC [6], Armada [29], Iris [24], CCAL [16, 17], and FCSL [34], among many others. These frameworks use a range of methods, such as movers [28] and concurrent separation logic [3]. Although there has been much recent progress in using these frameworks to verify shared-memory concurrent systems, handling concurrency still brings additional proof burden compared to verification of sequential systems. DaisyNFS's design isolates this verification overhead to the transaction system's proof, and then uses Dafny to reason about file-system operations. Furthermore, it would be challenging to extend a concurrency framework with crash safety compared to starting with Perennial, which required non-trivial extensions to add crash-safety support to Iris.

IronFleet [19] applies Dafny's sequential reasoning to a non-sequential setting, namely to verify event handlers for distributed systems. Each event handler is structured in phases: first messages are received, some local computation is done, and then messages are sent. This structuring enables a reduction argument [28] which makes it sound to treat each event handler as if it ran in an atomic step, with no interleaving of steps by other machines. Instead of a reduction argument, DaisyNFS uses the transaction system to make operations atomic. Although DaisyNFS operations may only access shared state through the transaction-system API, there are no phases or constraints on the ordering of reads and writes within a transaction.

2.3 Verified two-phase locking

Chkliaev et al. [11] verify serializability of two-phase locking and other transaction concurrency control mechanisms in the PVS theorem prover. Their proof formalizes two-phase locking as an abstract protocol consisting of sequences of read, write, and locking operations, as opposed to a concrete implementation as in DaisyNFS. Pollak [32] uses a variant of the CAP separation logic [15] to give a pencil-and-paper proof of serializability for a two-phase locking implementation.

Lesani et al. [27] developed a framework for verifying software transactional memory algorithms, modeled as I/O automata. They applied their framework to sophisticated STM algorithms, such as the NOrec algorithm [14]. The STM algorithms considered do not handle persistence and the framework does not address crash-safety reasoning.

2.4 Unverified file systems

We chose to verify an NFS server because it is widely used in practice and the expected behavior of NFS operations is well

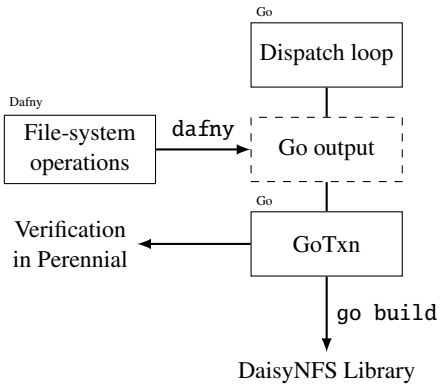


Figure 1: The structure of the code.

documented in RFCs. FUSE is an alternative for implementing file systems in user space, but its operations have a less clear specification.

Isotope [35] is a block-level transaction system similar to GoTxn in its API which was used to implement a file system called IsoFS. Its logging design is based on multi-version concurrency control (MVCC) [1] rather than our use of pessimistic locking. IsoFS has a similar design to DaisyNFS: it factors out isolation and atomicity to the transaction system, making it easy to handle crashes and concurrency. Unlike GoTxn and DaisyNFS, Isotope is unverified and thus prone to subtle concurrency bugs in the transaction system and bugs in the IsoFS code, whereas DaisyNFS uses the split design to verify both the transaction system and the transactions themselves.

To be conducive to verification, DaisyNFS is implemented differently than many NFS servers; the main differences are that using two-phase locking is not common practice, and most NFS servers are implemented on top of an existing file system. For example, the Linux NFS server can export any underlying file system supported by the kernel. An exported file system such as ext4 may use a journaling system, but the file system and VFS layers perform locking and are still prone to concurrency bugs. WAFL [21] is an NFS appliance that provides snapshots and logs NFS requests to NVRAM. It has evolved its locking plan to obtain good parallelism [13]. Both the Linux NFS server and WAFL are more complicated and have more features than DaisyNFS.

3 System design

As shown in Figure 1, DaisyNFS is implemented in three layers: 1) a dispatch loop that speaks the NFS wire protocol and calls the appropriate method for each operation; 2) a Dafny class that implements each method; and 3) a transaction system that applies the updates of each method to the disk atomically. The dispatch loop is unverified; we assume that the server correctly decodes messages, calls the right method for an operation, and encodes the response. The middle layer implementing the file-system operations is written and verified in Dafny, which has a backend for Go. The third layer is directly written in Go and verified using Coq and Perennial. By implementing the file system on top of the transaction system, we can implement each NFS method in Dafny as sequential code calling into a concurrent transaction system library. The NFS operations supported by DaisyNFS are listed in Figure 6.

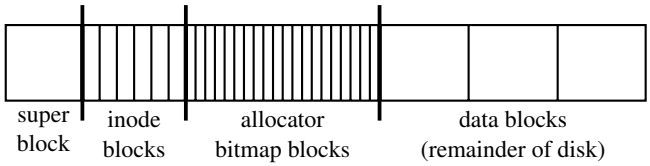


Figure 2: The layout of the file system on top of the transaction system's disk. The number of inode blocks and data bitmap blocks are compile-time constants, but easy to change without affecting the proofs.

3.1 Dafny file system

The file system is responsible for implementing files and directories onto an array of disk blocks that is exported by the transaction system. The disk layout used by the file system is shown in Figure 2, with regions for inode blocks, bitmap blocks, and data blocks for files and directories. This figure is in terms of the disk exported by the transaction system; the transaction system itself has a 513-block write-ahead log to support multi-block atomic writes to the disk.

The high-level organization of the file system separates three concerns, each building upon the previous: (1) implementing indirect blocks so files can be up to 512GB, (2) implementing byte-granularity reads and writes on top of blocks, and (3) implementing directories by encoding them as files with a special type together with operations to manipulate those files. §7 explains the internals of the file-system design in more detail, alongside the structure of the Dafny proof.

3.2 Transaction system

The transaction system handles concurrency and crash safety, and its API is listed in full in Figure 3. The file system creates an empty transaction by calling `Begin()`. The entire transaction appears to execute atomically when the caller finishes with `Commit`, or the transaction is discarded with no effect on `Abort`. Reads and writes operate on addresses which specify a position by giving a block number and an offset in bits (always less than $4096 \cdot 8$, the number of bits in a block). The `Read` method requires an explicit size argument while the size of a `Write` is implicit in the size of the `data` slice. We separate out the bit-sized operations to `ReadBit` and `WriteBit` (rather than using a single-element byte slice) to simplify the specification.

Figure 3 also shows the allocator API alongside the transaction API because its implementation is part of the interface that the Dafny code has access to. Allocation does not behave atomically along with the rest of the transaction, which the proof handles by allowing allocation to return any value. In

```
type Addr struct {
  Blkno  uint64
  Offset uint64
}

// starting and stopping a transaction
func Begin() *Txn
func Abort(tx *Txn)
func Commit(tx *Txn)

// operations within a transaction
func Read(tx *Txn, a Addr, sz uint64) []byte
func ReadBit(tx *Txn, a Addr) bool
func Write(tx *Txn, a Addr, d []byte)
func WriteBit(tx *Txn, a Addr, d bool)

// allocator API
func NewAllocator(max uint64) *Allocator
func Alloc(a *Allocator) uint64
func Free(a *Allocator, n uint64)
```

Figure 3: The API for the transaction system and allocator, both of which are available within the Dafny file-system implementation. Reads and writes between `Begin` and `Commit` appear to execute atomically on disk and for other threads, while `Abort` guarantees the transaction has no effect. The allocator's `Alloc` and `Free` operations are safe to call concurrently.

practice the way the file system uses such a non-deterministic specification is to store the ground-truth allocation state in the transaction system, and then to use the allocator as a hint to find free bits. As a result the return value of `Alloc()` must be checked against the durable bitmap with `ReadBit()`. Similarly, to free an address it must be both freed in memory and on disk with `WriteBit()`.

The transaction system builds upon GoJournal, verified in prior work [9], adding two-phase locking on top to implement transactions. While a transaction is running, it acquires locks for any addresses it reads or writes, and on abort or commit, it releases all locks held. Transactions that don't conflict can prepare in parallel, and GoJournal will batch concurrently committed transactions for efficiency.

Acquiring multiple locks during a transaction creates the possibility for deadlocks, for example if two threads acquire a pair of locks in the opposite order. The two-phase locking implementation does not implement a specific lock acquisition order, leaving it to the file system to avoid deadlock — the most interesting case is `RENAME`, which is discussed in more detail in §7.1.1.

4 Specifying DaisyNFS

The specification for DaisyNFS is a state machine describing an ideal NFS server in the form of an abstract state and a transition for each operation. The implementation of DaisyNFS is a binary `daisy-nfsd` that implements the NFS protocol, running on top of a disk. Then the DaisyNFS correctness theorem is a *refinement* property, which intuitively says that for any interaction with the implementation, the ideal, atomic NFS state machine could produce the same responses; §4.2 gives a more formal definition. As a result a client interacting with the server can pretend that it is the NFS state machine and ignore the complexities of its implementation.

4.1 Formalizing NFS

RFC 1813 specifies the NFS protocol, which we make mathematically precise with a state-machine representation defined in Dafny. The formalization requires first defining what state operations modify, and then a transition for each NFS operation that specifies how it changes the state and what return values are allowed. While most of the specification is deterministic, some operations have to be specified with non-determinism; for example, we allow returning an out-of-space error in many operations, and the specification allows any timestamp to be picked for the current time. The RFC is precise about arguments and allowed return values, and the text is good about explaining the intended behavior, but it does not describe the state an NFS server maintains. We define the NFS server state as shown in Figure 4.

```
// the abstract state of the file system
type FilesysData = map<Ino, File>

datatype File =
  | ByteFile(data: seq<byte>, attrs: Attrs)
  | Dir(dir: map<FileName, Ino>, attrs: Attrs)

type Ino = uint64
type FileName = seq<byte>
datatype Attrs = Attrs(mode: uint32, ...)
```

Figure 4: Dafny definition of the NFS server state (simplified).

This definition says that an NFS server conceptually maintains a mapping from inode numbers to files, where a file can either be a regular file with bytes, or a directory. Both types of files have a number of attributes, storing metadata like the file's mode (permission bits) and modification time. A directory is a partial map from file names (which are just bytes) to inode numbers. Note that DaisyNFS doesn't represent the file system as a tree but as a collection of links, which is sufficient to model all NFS operations, because NFS clients resolve pathnames.

The NFS state machine models each operation as a non-deterministic transition, written as a predicate that holds when it is allowed for an operation to change the state from `fs` to `fs'` and return `r`. The return value is always wrapped in a `Result` type, which can be either `Ok(v)` for a normal return or an error code for one of the errors defined in the standard. We systematically guarantee that the state is unchanged when an operation returns an error (though this is stronger than what the RFC requires); the transaction system makes this easy to achieve by aborting the whole transaction. For example, Figure 5 shows the specification for a (hypothetical) `GETSZ` operation that returns the size of the inode `ino`.

There are four clauses in the specification. The first just says that this operation is read-only. The second is one possi-

```
predicate GETSZ_spec(ino: Ino, fs: FilesysData,
  fs': FilesysData, r: Result<uint64>)
{
  fs' == fs &&
  (r.ErrBadHandle? ==> ino !in fs) &&
  (r.ErrIsDir? ==> ino in fs && fs[ino].Dir?) &&
  (r.Ok? ==> ino in fs && fs[ino].ByteFile? &&
            r.v == |fs[ino].data|)
}
```

Figure 5: Specification of a hypothetical `GETSZ` operation, a simplification of the real `GETATTR` operation.

Category	Operations	Verified
File and directory ops	`GETATTR`, `SETATTR`, `READ`, `WRITE`	✓
	`CREATE`, `REMOVE`, `MKDIR`, `RENAME`	✓
	`LOOKUP`, `READDIR`	✓
Unsupported features	`READLINK`, `SYMLINK`, `LINK`, `MKNOD`	✗
	`READDIRPLUS`, `ACCESS`	✗
Configuration	`FSINFO`, `PATHCONF`, `FSSTAT`	✗
Trivial operations	`NULL`, `COMMIT`	✓

Figure 6: NFS API and which operations DaisyNFS supports and verifies.

ble error: if the server returns `ErrBadHandle`, then `ino` is not allocated. The third is a different error, which says this operation returns `ErrIsDir` for directories. Finally the fourth case says that if the operation is successful, it returns the length of the data in `fs[ino]`. Dafny checks several consistency properties of this specification itself; for example, a use of `fs[ino]` will not even compile if the specification does not earlier imply `ino in fs`.

We developed a state-machine model of the regular file and directory operations in NFS in this style, including specifying what certain errors signify. Figure 6 lists the entire NFS API and what parts we verified.

DaisyNFS implements `FSINFO` and `PATHCONF`, which give the client static configuration information about the file system (for example, the maximum supported write size). These return constants and thus have no specification. DaisyNFS also implements `FSSTAT` to report total and free space, but it does not have a meaningful specification.

DaisyNFS could support some of the remaining operations with some more effort. Support for symlinks and `MKNOD` would require mostly mechanical changes to accommodate new file types. `LINK` is more complicated because in addition to tracking the link count of every file in the state, the specification for `REMOVE` needs to say that the link count is decremented and that the file is deleted if its link count drops to zero.

4.2 Specifying correctness for DaisyNFS

The transition system in §4 describes the abstraction of an NFS server, but what does it mean for the `daisy-nfsd` binary to implement this specification? To formalize DaisyNFS's correctness we use a definition of *concurrent, crash-safe refinement*, which informally says that every execution of that server binary — including with concurrent operations and crashes — has user-visible behavior that the specification could also produce (that is, the behavior is allowed by the specification). In DaisyNFS's specification the visible behavior is defined to be network requests and responses.

To define the specification, we need to be more precise about what a program is and how it executes, since these programs are used to model the DaisyNFS code and specification. We write $p : \texttt{Go}\langle \text{X} \rangle$ to say p is a Go program written using operations from layer X, where X is one of NFS, Txn, or Disk. Layer operations are always atomic transitions in a state machine. In the NFS layer, the operations behave according to the NFS state machine described previously in §4.1 and defined formally in Dafny. The Txn layer is specified both in Coq where it is part of the transaction system's correctness theorem and in Dafny where it appears as an assumption. The Disk transition system is formalized in Coq as part of the GoJournal proof, and assumes reads and writes of 4KB blocks are atomic. Each layer includes concurrent threads that interleave layer operations, basic heap operations on pointers, slices, and maps, and computation on primitives like integers and structs.

The correctness of DaisyNFS is stated in terms of a program that repeatedly receives a request, processes it in a background thread, and sends a response, which is intended to model the core behavior of the `daisy-nfsd` server. A schematic depiction of this server loop is given in Figure 7. This code starts by recovering the state of the system on line 3. Then it repeatedly accepts new requests from the network, abstracted with `GetRequest()` (including parsing the NFS wire protocol). These requests are each processed in a background thread due to the goroutine spawned on line 6. The processing for each request dispatches to the appropriate file-system operation (e.g., lines 9 and 12). The implementations of these operations are compiled from Dafny to Go and then linked with the transaction system.

The correctness theorem references three versions of this loop, at different levels of abstraction. At the top, the specification is a loop $\texttt{s}_{\text{NFS}} : \texttt{Go}\langle \text{NFS} \rangle$ which atomically processes each NFS operation according to the NFS state machine.

Below the NFS layer, $\texttt{s}_{\text{dfy}}$ models the server where each operation is replaced with its Dafny implementation, wrapped in a transaction. In this layer we write $\texttt{atomically}\{\texttt{f}\}$ to represent a transaction running `f`, which by definition in the Txn layer runs atomically for specification purposes. An `atomically` block corresponds to executable code that follows a pattern like `tx := Begin(); f(tx); tx.Commit()` to run `f` in the context of a GoTxn transaction (some additional code handling aborts is omitted in this snippet).

The final layer that models the executable code is given using a function $\text{link}(p, i)$, which takes a program p using operations from layer S and substitutes each operation with an implementation according to $i : S \to \texttt{Go}\langle \text{T} \rangle$. The notation "link" is intended as an analogy to the linking phase of compilation, taking a program p with some undefined symbols

```
// this is the core of daisy-nfsd
func main() {
  fs := filesys.Recover()
  for {
    req := GetRequest()
    go func() {
      switch req.Op {
      case CREATE:
        ret := fs.CREATE(req.Args)
        SendReply(req, ret)
      case LOOKUP:
        ret := fs.LOOKUP(req.Args)
        SendReply(req, ret)
        // ... other cases ...
      }
    }()
  }
}
```

Figure 7: A schematic depiction of the server loop, written in Go. $\mathtt{s}_{\mathrm{NFS}}$ looks like this code, but by definition all operations (for example the calls to `fs.CREATE` and `fs.LOOKUP`) are processed atomically and according to the NFS transition system. As far as the proof goes `GetRequest()` and `SendReply()` just produce a trace of I/O behavior and are unverified.

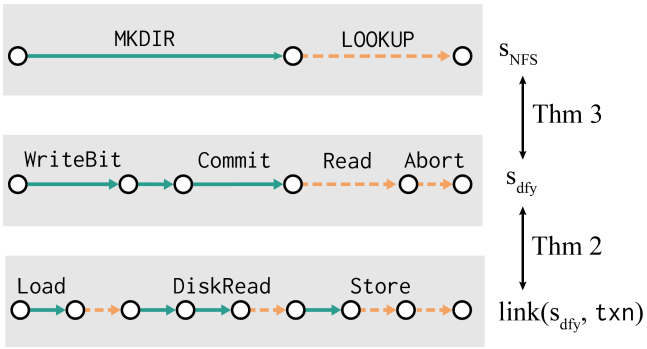


Figure 8: Illustration of the DaisyNFS proof strategy in terms of one possible execution of DaisyNFS, receiving parallel `MKDIR` and `LOOKUP` operations, at its three abstraction levels. Operations in each row are coded green and solid or orange and dashed according to which operation they correspond to (the top-level `MKDIR` or `LOOKUP` respectively). The refinement proof first shows that for every code execution (bottom row), there exists an atomic execution at the Txn layer (middle row), as proven in Theorem 2. This justifies sequential reasoning to show the transactions follow the NFS specification (top row), as proven in Theorem 3. Finally Theorem 1 puts the two together.

and substituting each symbol s with a call to an implementation of that method given by the library code $i(s)$. We write $\mathrm{link}(\mathtt{s}_{\mathrm{dfy}}, \mathtt{txn})$ to represent linking the Dafny code with the transaction system's implementation `txn`.

The proof is about the server loop at the core of `daisy-nfsd` at three layers of abstraction. Figure 8 illustrates one execution of the DaisyNFS server where two clients issue `LOOKUP` and `CREATE` in parallel, at the three levels of abstraction: the bottom shows an execution of $\mathrm{link}(\mathtt{s}_{\mathrm{dfy}}, \mathtt{txn})$ at the Disk layer, the middle a corresponding atomic execution of $\mathtt{s}_{\mathrm{dfy}}$ at the Txn layer, and finally the top-level has a single transition for each operation at the NFS layer.

Refinement relates two programs in terms of their visible behavior, which we will use to connect the server loop at the disk layer to the transaction layer and finally to the NFS layer. For the purposes of this paper, all of the programs involved are servers that issue network I/O, either receiving an NFS request or responding to one. Regardless of the level of abstraction, each model of the server defines a trace of network I/O consisting of requests and responses, and this is the behavior refinement talks about:

Definition (Concurrent, crash-safe refinement)**.** An implementation program p_c is a *concurrent, crash-safe refinement* of a specification program p_s, written $p_c \sqsubseteq p_s$, if whenever there are initial states σ_s and σ_c satisfying $\mathrm{init}(\sigma_s, \sigma_c)$ and p_c can execute from σ_c and produce a trace of network I/O tr, then p_s can execute from σ_s and produce the same trace tr. Execution might involve crashing and restarting a program (potentially multiple times), wiping out any in-memory state after each crash. When we state $p_c \sqsubseteq p_s$ we leave implicit the definition of initial states $\mathrm{init}(\sigma_s, \sigma_c)$, which will generally say both states are all zeros and of the same size.

The intuition behind the notation $p_c \sqsubseteq p_s$ is that the set of behaviors of p_c (the set of traces of network I/O tr) is a subset of the behaviors of p_s.

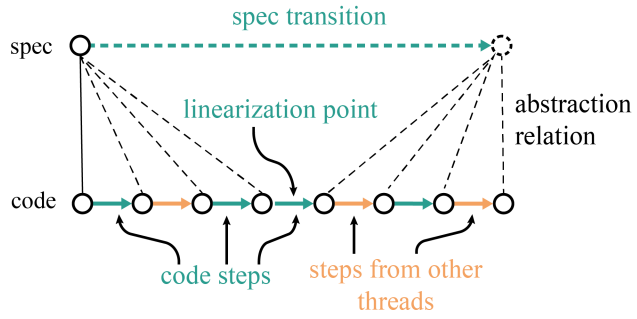
Now we have enough to state the final DaisyNFS correctness theorem:

Theorem 1 (DaisyNFS correctness)**.** $\mathrm{link}(\mathtt{s}_{\mathrm{dfy}}, \mathtt{txn}) \sqsubseteq \mathtt{s}_{\mathrm{NFS}}$.

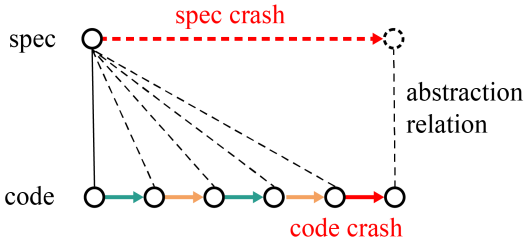
In this correctness theorem, initialization requires running a Dafny method on an empty disk. Subsequently the system boots by first recovering the transaction system, then restoring the file system. Theorem 1 will follow from the correctness of the transaction system combined with the results from Dafny.

5 Verification approach

DaisyNFS's concurrent, crash-safe refinement is a much more sophisticated property to verify than sequential refinement. Figure 9 illustrates the complexity of proving a concurrent and crash-safe refinement, whereas Figure 10a shows the relatively simple per-operation obligation for sequential reasoning. For both forms of refinement, the basic proof technique is to construct a *forward simulation* from the code execution to the specification transition system, which requires an abstraction relation connecting their states and a proof that shows the abstraction relation is preserved by operations. In a sequential, non-crash simulation, it is sufficient to show that each operation restores the abstraction relation when it returns since its intermediate states are invisible. The complication in a concurrent simulation is that the code can have many concurrent threads, each running a different operation at the specification level. The proof of any given operation must also show that the intermediate states satisfy the abstraction relation, since at any time other threads might run. Similarly, the proof of each operation's implementation must consider interference with its execution from other threads at any time.



(a) Linearizability obligation (for each operation)



(b) Crash-safety obligation (for each operation)

Figure 9: Obligations for verifying a concurrent, crash-safe refinement. The proof of refinement must show that every operation simulates the abstract specification for the operation at some *linearization point* (as illustrated in 9a), and that a crash simulates a specification crash transition (as illustrated in 9b). The abstraction relation must be preserved at all intermediate points, including after a crash.

5.1 Simulation transfer

The design of DaisyNFS uses transactions, and in particular GoTxn, to simplify the proof of concurrent refinement. Transactions appear to run sequentially, and thus should permit reasoning about the body of each transaction sequentially even though the actual execution interleaves multiple transactions. A key contribution of this paper is the formalization of a *simulation-transfer theorem* which proves that a system implemented with transactions that is verified with a *sequential* forward simulation against some specification refines the same specification in the sense of a *concurrent, crash-safe refinement* when run through GoTxn.

Due to simulation transfer, we can use the simpler verification methodology of sequential simulation for the DaisyNFS file-system code, compared to the Perennial program logic used to verify the transaction system underneath. To fully take advantage of this difference, DaisyNFS is verified using Dafny [26], an entirely different tool. Dafny is a verification-oriented programming language that is restricted to sequential proofs. The use of Dafny greatly reduces the proof burden for verifying DaisyNFS, because sequential proofs are well-suited to automation and Dafny's automation is well-developed (in contrast automation for concurrent proofs is still nascent, and would need to be integrated into Perennial to be used for these proofs).

The value of sequential proofs can be seen in the proof-to-code ratio for the transaction system, which is around 20×, versus the Dafny proofs which required about 2× as many lines of proof as code. Further evidence can be seen in the incremental development of DaisyNFS, which §9.4 further elaborates on.

To make simulation transfer this precise, let us first define "sequential reasoning" more formally. Suppose we have an implementation of layer S using operations from T. Note that all the proofs about the transaction system are for an arbitrary system with operations in S; though we use the system with an implementation of NFS, the GoTxn proof is more general. The implementation i consists of a function $i(op) : \texttt{Go}\langle \text{T} \rangle$ for each operation $op \in S$. The statement seq_refinement$\langle T, S \rangle(i)$ says that i is a correct sequential implementation of S using T. To specify the normal behavior of each operation, the definition refers to $s \overset{op}{\rightsquigarrow} s'$, which says op can transition from s to s' according to the definition of layer S. To specify correctness under crashes, this definition refers to crash(t,t') and crash(s,s'), which are the crash transitions for layers T and S respectively and model, for example, clearing the contents of memory.

Definition (Sequential refinement)**.** The implementation $i : S \to \texttt{Go}\langle \text{T} \rangle$ is a *sequential refinement*, written seq_refinement$\langle T, S \rangle(i)$, if there exists an abstraction relation $R \subseteq \Sigma_S \times \Sigma_T$ such that:
(1) for every operation $op \in S$, the following sequential Hoare triple holds:

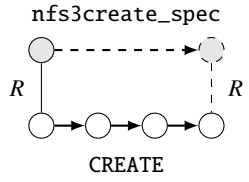
$$\{\lambda t.\, R(s,t)\}\ i(op)\ \left\{\lambda t'.\, \exists s'.\, R(s',t') \wedge s \overset{op}{\rightsquigarrow} s'\right\},$$

(2) init(s,t) must imply $R(s,t)$, and
(3) if $R(s,t)$ and crash(t, t') hold, then there exists an s' such that $R(s',t')$ and crash(s,s').

Conditions (1) and (2) in this definition are standard for sequential verification of refinement, while condition (3) is a standard condition for sequential crash-safety [7]. Though condition (3) requires the abstraction relation to be preserved by crashes, the proof engineer does *not* have to reason about crashes in the middle of operations. The diagram in Figure 10 depicts the main refinement condition (1) diagrammatically.

Simulation transfer takes a proof of *sequential* refinement conditions for a system implemented using transactions and derives a *concurrent and crash-safe* refinement. A transaction must satisfy some conditions to ensure atomicity. We write safe(p) to say that p is a valid transaction. The main restriction is that p cannot access global state such as the heap, since the transaction system does not make such accesses atomic. The implementation i in this theorem gives only the body of each transaction; the theorem instead references atomically $\circ\, i$ where $(\text{atomically} \circ i)(op) = \texttt{atomically}\{\, i(op)\,\}$ uses the macro from the Txn layer to specify that the operation is wrapped in a transaction and is thus by definition atomic.

Theorem 2 (Simulation transfer)**.** Let S be a spec layer implemented using transactions with $i : S \to \texttt{Go}\langle \text{Txn} \rangle$, such that



(a) Obligation for sequential refinement.

```
method CREATE(d_ino: uint64,
              name: Bytes)
 returns (r: Result<Ino>)
 requires R(txn_disk, fs)
 ensures R(txn_disk, fs)
 ensures r.Ok? ==>
 nfs3create_spec(d_ino, name,
   old(fs), fs, r.v)
```

(b) Dafny encoding

Figure 10: Illustration of seq_refinement(i_{NFS}) (left) and its encoding in Dafny seq_refinement$_{\text{dfy}}(i_{NFS})$ (right), for one particular operation. In the diagram, the solid parts are assumed, and the dashed parts must be shown to exist. The complete Dafny spec is more precise about errors.

seq_refinement(i) and $\forall op.\,\text{safe}(i(op))$ hold. Then

$$\forall p : \texttt{Go}\langle \text{S}\rangle, \text{link}(\text{link}(p, \text{atomically} \circ i), \texttt{txn}) \sqsubseteq p.$$

Simulation transfer says that if an implementation of S using transactions is correct in a sequential sense, then this is sufficient for any spec program p to have atomic and correct behavior for its primitives when run with GoTxn. The executable code for p derived in two steps: $\text{link}(p, \text{atomically} \circ i)$ replaces the operations in S with their atomic implementations at the GoTxn API level, while $\text{link}(\text{link}(p, \text{atomically} \circ i), \texttt{txn})$ takes the result of this process and substitutes the actual GoTxn implementations of `Begin`, `Read`, `Commit`, and so on. In §6 we discuss how this theorem is proven using Perennial and Coq.

5.2 Putting simulation transfer together with Dafny proofs

In order to use simulation transfer to obtain Theorem 1, we need to prove that DaisyNFS's implementation, i_{NFS}, satisfies the sequential refinement conditions. To do so, we define seq_refinement$_{\text{dfy}}(i)$, an encoding of sequential refinement using Dafny pre- and post-conditions (as illustrated in Figure 10), and prove that DaisyNFS satisfies these conditions in Dafny. The crash refinement condition (3) is straightforward; crashes have no effect in both the Txn layer and the NFS layers because they do not have ephemeral state. Details on how the Dafny obligations handle initialization and recovery are found in the first author's thesis [5: §6.4].

Lemma 3. seq_refinement$_{\text{dfy}}(i_{NFS})$ holds.

From here we can apply Theorem 2 to Lemma 3 and obtain Theorem 1, which says $\text{link}(\texttt{s}_{\text{dfy}}, \texttt{txn}) \sqsubseteq \texttt{s}_{\text{NFS}}$ (note that $\texttt{s}_{\text{dfy}} = \text{link}(\texttt{s}_{\text{NFS}}, \text{atomically} \circ i)$). Figure 8 illustrates just one execution that the theorem covers: the transaction system proof guarantees an atomic execution while the sequential refinement guarantees the transactions themselves are correct. There are two trusted assumptions needed for the theorems to compose. First, seq_refinement$_{\text{dfy}}(i_{NFS})$ should imply seq_refinement(i_{NFS}). That is, the encoding of the refinement conditions in Dafny must be correct, but also the semantics of the transaction system operations modeled in Dafny must match the Coq proof. Second, every Dafny transaction must be valid, meaning safe($i_{NFS}(op)$). The Dafny code satisfies safety due to a simple syntactic check: the only mutable state in the file-system Dafny class is the transaction system, so file-system operations cannot make mutations other than through GoTxn.

6 Verifying the transaction system

This section describes the implementation and proof of the transaction system, GoTxn. A contribution of this paper detailed in this section is to verify the powerful specification of Theorem 2 on top of a real implementation, which required verifying two-phase locking using local reasoning in Perennial unlike the more typical textbook proofs that reason about the global execution of many concurrent transactions. Note that this section is only about the transaction system and has nothing specific to the file system implemented on top.

6.1 GoTxn's implementation

GoTxn is implemented as an extension to GoJournal [9], a journaling system verified in Perennial. The journaling system provides the ability to write multiple objects atomically, with an implementation that provides good concurrency. For correctness GoJournal relies on the caller to guarantee that concurrent operations do not access the same disk objects. GoTxn automatically provides the concurrency control to guarantee this precondition using two-phase locking (2PL). The result is an interface that behaves atomically without any concurrency reasoning from the caller.

The two-phase locking system logically maintains a lock per object. The algorithm gets its name from an *expanding* phase in which reads and writes acquire locks as needed, followed by committing the transaction's writes to the journal and a *contracting* phase where all the acquired locks are released. Instead of committing, a transaction can abort early to abandon buffered writes and release the locks acquired so far, in which case the disk is unaffected. The whole operation appears to execute atomically at commit time; reads return their results early, but the locks ensure these values remain consistent up until the commit point. The GoTxn proof makes the informal correctness argument precise by giving a proof of a refinement-based specification.

6.2 Verifying two-phase locking with local reasoning

In §5.1, we gave Theorem 2 as the specification for the transaction system. Recall that this theorem converts sequential refinement proofs for transactions into concurrent refinement. To prove this, we first use Perennial to show that code encapsulated in a transaction truly behaves atomically, formalized with the following theorem:

Theorem 4. The GoTxn implementation `txn` is a *transaction refinement*, meaning for all $p : \texttt{Go}\langle\text{Txn}\rangle$ where $\text{safe}(p)$ holds, $\text{link}(p, \texttt{txn}) \sqsubseteq p$. The definition of $\text{init}(s,t)$ in this refinement relates an all-zero physical disk to an all-zero transactional disk of the same size.

Theorem 4 captures the intuition that transactions provide atomicity, while Theorem 2 formalizes why atomicity provides sequential reasoning. The proof of Theorem 2 from Theorem 4 is conceptually straightforward. Since the `atomically` blocks in p ensure transaction operations run without interruption, the sequential refinement diagram can be applied to code inside these blocks.

The proof of Theorem 4 itself in Perennial is more involved. The high-level approach is to encode refinement as Perennial Hoare triples, one for each operation [8, 38]. To make this sound for concurrent refinement, (1) the proof must identify and verify the *linearization point* of an operation, the time at which the operation appears to have executed; and (2) the proof tracks logical *ownership* of state, and threads may only modify state that they have "acquired" ownership of through synchronization. The resulting proof style is called "local" because we reason about each thread in isolation, considering just the parts of state it accesses. Using Perennial enables us to re-use the existing GoJournal proof, but this local proof-style is quite different from standard proofs of serializability for two-phase locking, which reason globally about the set of transactions and ordering constraints imposed by locks.

In more detail, the refinement proof must show that the code `tx := Begin(); f(tx); tx.Commit()` has a subset of executions of the `atomically{f}` construct. The difficulty in proving this is that the linearization point is at the very end when the code calls `Commit`, at which point the actual earlier execution of `f` becomes visible to other threads. We must argue that at this point the entire `atomically{f}` block's effect has occurred by tracking the behavior of `f`.

As the transaction executes, the proof tracks the initial value of any objects accessed in a map J. The domain of this map $\Sigma = \text{dom}(J)$ is the *footprint* of the transaction, which two-phase locking keeps locked during the transaction. The intuition behind the invariant is that if the transaction only depends on J, the transaction's execution can be delayed to take place atomically at the call to `Commit`, because locking prevents the subset J of the journal from being accessed by other threads. In particular the proof sets up a set of *lock invariants* that say the lock for address a is needed to access the GoJournal resource $a \mapsto_d o$, which gives permission to read and write to a. See the thesis for a more formal connection to the GoJournal specification [5: §5.5].)

The proof maintains a refinement relation during the execution of a transaction `f`, which is formally expressed using the GoJournal resources but explained more intuitively here. Let J be a map with the values of each object in the transaction's footprint Σ at the first time they are accessed by `f`, and let J' be a map with the transaction's current buffered in-memory view of the same addresses. Then, the invariant requires that after n steps of execution:

1. The transaction holds the lock for every address $a \in \Sigma$.
2. Executing n steps of `f` in *any* starting state that has the same values as J for the addresses in Σ can lead to a state with values given by J'.

At the start of a commit, the locking described by the first part of the invariant ensures that the durable value of each address still match the value in J, and is required to call the GoJournal `Commit` operation. The second part of the invariant means that even though other parts of the state outside of Σ may have changed, those changes do not affect the execution of `f`. Thus, executing `f` at this point in a single step would have the same behavior as the implementation has observed. The GoJournal `Commit` specification ensures that the durable values of objects in the footprint are atomically updated to match J'.

Showing that the second part of the invariant holds requires that code within a transaction must not access global state outside of the transaction system, as mentioned at the end of §5.1. Accesses to such global state would violate the invariant because their behavior would then depend upon things outside of the footprint Σ. Because those global values could change by the time the transaction commits, the above argument would no longer work if they were allowed.

The allocator creates another subtlety related to the second part of this invariant. Allocations do not hold the allocator lock throughout the remainder of a transaction. This seems to violate the two-phase locking pattern, since allocations could be implicitly observed by other concurrent transactions from the fact that an allocated address is no longer free. Correspondingly, in the proof, the footprint J of a transaction does not describe the allocator state. Thus, at the linearization point, the addresses returned by the allocator may no longer be free. However, because the specification for the allocator does not guarantee that returned addresses are actually free, the second part of the invariant above still holds.

7 Verifying the Dafny implementation

We follow the standard approach for verifying software in Dafny: each file-system operation is implemented as a method on a class and its specification is given using pre- and post-conditions. §5.1, explains how the Dafny proof shows the code is a correct implementation of NFS in terms of sequential refinement. This section provides details about the file-system design and proof.

DaisyNFS is implemented and verified in several layers of abstraction, depicted in Figure 11. Each layer is implemented as a class that wraps the lower layer as a field. The transaction system is an assumed interface in Dafny, while the complete server implements the NFS wire protocol and calls into the top-level Dafny class for each operation.

Layer	Functionality
dir	Directories and top-level NFS API.
typed	Inode allocation.
byte	Implement byte-level operations using blocks.
block	Gather blocks for each file into a single sequence.
indirect	Triple-indirect blocks organized in a tree.
inode	In-memory, high-level inodes; block allocation.
txn	Assumed interface to GoTxn.

Figure 11: Layers in the Dafny implementation and proof of the file-system operations.

Between the layers of the file system there are three difficult pieces of functionality: organizing data blocks into metadata and data (the indirect and block layers), translating byte-level operations into block operations (the byte and typed layers), and implementing directories as special files that the file system itself reads and writes (the dir layer). The modularity was essential to complete the proof in manageable chunks (to avoid overwhelming the developer and prover), and it would have been natural even without verification.

7.1 Implementing the file system using transactions

The design of DaisyNFS is broadly similar to the file system in xv6 [12], as well as Yggdrasil [36], a verified sequential file system. We also adopt the recursive strategy for implementing and verifying indirect blocks from DFSCQ [25]; recursion simplifies the implementation of triple-indirect blocks, which are needed to reach a reasonable maximum file size of 512GB. Unlike most file systems, DaisyNFS is designed to fit every operation into a transaction in order to support our goal of sequential reasoning. This is a non-standard design and we encountered some unique challenges in doing so. In this section we highlight difficulties in fitting two features into transactions: renaming and freeing space from deleted files.

7.1.1 Avoiding deadlock in renames

The NFS RENAME operation is similar to the `rename` system call: it moves a source file or directory to a destination location. What makes it tricky is that it involves more than one inode and hence introduces the possibility for deadlock. We use the standard strategy of enforcing a global ordering where inodes are always locked in numerical order (smaller inode numbers first); this avoids a deadlock where a cycle of threads is waiting on each other.

In a rename operation, the source and destination are each specified by a combination of the parent directory inode and name within that directory. Rename has an additional functionality of overwriting the destination if the source and destination are files, or if both are directories and the destination is empty. It is this latter check that makes deadlock avoidance difficult: it is necessary to lock the source and destination directories first to lookup the source and destination names, but those might be files that are earlier in the inode lock order. We address this in the code by returning an error from the Dafny transaction before the lock order would be violated. The error comes with the set of inodes that should have been acquired. The rename is then re-run with this set of inodes as a lock hint; these are first acquired in the correct order, then compared against the current source and destination in case they have been renamed concurrently.

At this point it is worth discussing the performance considerations that lead to handling lock ordering in the file system, rather than generically in GoTxn. The transaction system could avoid deadlocks by either enforcing a global order over addresses or by timing-out operations. Enforcing a global order is inefficient for the file system; data blocks will never cause deadlock because the file system only accesses a block after locking the (unique) inode that owns it. Timing-out operations would lead to slow and spurious transaction failures that could more rapidly be avoided in the higher-level code, hence we do not attempt to detect deadlock dynamically.

7.1.2 Freeing space

Freeing space becomes surprisingly tricky with large files. The problem is that a large-enough file may reference too many blocks to be freed in a single transaction. DaisyNFS handles freeing by removing a file from its directory and marking it free in one transaction, and in separate transactions reclaiming the space it took by deallocating its blocks.

Removal is implemented as a combination of two transactions, one which performs the logical operation but leaks space, and an operation `ZeroFreeSpace(ino)` which frees and zeroes the unused space in an inode that we prove has no effect on the file-system state. Because this operation is a logical no-op, it is safe to call it at any time. In practice the implementation is careful to call it after any operation that leaves unused blocks, in particular `SETATTR`, which can shrink a file by reducing its size, and `REMOVE`, which deletes a file. Furthermore since `ZeroFreeSpace` doesn't affect the user-visible data, it may return early to avoid overflowing a transaction, which GoJournal limits to 511 blocks.

There is one case where freeing blocks is important for correctness and not just to reclaim space. Growing a file is supposed to logically fill the new space with zeros. If the file had old data in that space, it would not be zero but some previously written and deleted data, which both violates the specification and is a potential security risk. The way we handle this with background freeing is with a run-time check: when the `SETATTR` operation grows a file checks, it checks if the free space is already zero first, and if not fails with a special error code. The unverified code interprets this as a signal to immediately call `ZeroFreeSpace` and try the operation again. The same support also handles holes created by writing past the end of a file, which are similarly supposed to be zero.

The freeing implementation is an interesting example of using validation in verification. The specification for much of the freeing code is loose, allowing any data to be written

	proof	code	spec
GoJournal	29,000	1,419	
Transaction system	10,000	250	932 (Thm 2)
File system	6,787	4,051	630 (Thm 3)
Trusted interfaces	—	—	558
`daisy-nfsd`	*unverified*	1,144	—

Figure 12: Lines of proof, code, and trusted specification. GoJournal is included only for comparison; its specification is subsumed by the transaction system's.

to the free space. We only needed a strong specification for the code that checks if the zeroing is done; the rest of the code needs to be correct for this check to ever succeed, but we aren't required to prove it.

7.2 Achieving good performance

An important aspect of the Dafny proof was to write code in a way that produces high-performance Go code. Compared to Dafny's C# backend, the generated Go code for Dafny's built-in immutable collections has much additional pointer indirection and defensive copying. Using these data structures for byte sequences would simplify proofs, but has unacceptably poor performance in Go.

To avoid this performance problem we use an axiomatized interface to Go byte slices (`[]byte` in Go) whenever raw data is required, including file data and paths, and then modify these slices in-place. It was possible to axiomatize this API without any changes to Dafny; we use a standard Dafny feature of `extern` classes to specify a Dafny class `Bytes` in terms of ghost state of type `seq<byte>` but then implement it as in Go as a thin wrapper around the native `[]byte` type. This API is trusted, so we test it: for example to catch off-by-one errors in the specification, we wrote tests like `[]byte{1,2,3}[2]` and ran them in Go and (equivalent) Dafny.

The on-disk data structures—inodes, indirect blocks, and directories—are represented in memory in their serialized form and modified by updating this representation directly, avoiding copies to move between representations. These were first written with slower purely functional code, which was then migrated to imperative code that used the functional code as a specification.

Dafny's default integer type `int` is unbounded and compiled to big-integer operations. We used Dafny's `nativeType` support to instead define a type of 64-bit integers (that is, natural numbers less than 2^{64}) and compile this to Go's `uint64`. This requires overflow reasoning, but automation makes this palatable in the proof and the performance gain is significant.

8 Development effort

We implemented DaisyNFS in a combination of Go and Dafny, with proofs in the Perennial framework (which is a library in the Coq proof assistant, heavily based on Iris [23]) and inline in Dafny. The Go side uses GoJournal, which we extend with a transaction system and concurrent allocator. The implementation is publicly available.[1]

The lines of proof, code, and specification for the layers of the system are summarized in Figure 12. GoJournal is prior work but included for comparison purposes. The GoTxn correctness proof, Theorem 2, is relatively large because code executed in `atomically` blocks can include many Go operations modeled by Perennial, and the proof has cases to handle each operation. However the result of the proof is a relatively concise specification as a plain Coq statement that doesn't refer to the Perennial logic.

The file-system operations are implemented in Dafny, which helped us verify a relatively complete system without too much tedium. The proof-to-code ratio (where code is the number of lines extracted by Dafny's `/printMode:NoGhost` flag) is about $2\times$ for the file system code. The proof summarizes the implementation well, with about $1/7$th as many lines of specification as code (about half that specification is quite verbose and concerns error codes and attributes). For efficiency, the Dafny code has trusted interfaces to primitives like byte slices and integer-to-byte encoding. Together these are written in 558 lines of trusted Dafny code. Finally, to complete the NFS server required around 1,000 lines of Go code, about half of which bridge between the Dafny method signatures and the actual NFS structs.

Similar to VeriBetrKV [18], we followed a discipline of identifying and addressing timeouts in the proof. As a result, the overall build is fast: compiling the proofs takes only 12 minutes on a slow machine in continuous integration and 4 minutes on a laptop using eight CPU cores.

9 Evaluation

In this section we evaluate DaisyNFS along the dimensions of performance (§9.1 and §9.2), correctness (§9.3), and ease of change (§9.4).

9.1 Performance

To evaluate the performance of DaisyNFS, we ran three benchmarks: the LFS smallfile and largefile benchmarks, and a development workload that consists of `git clone` from a local repository followed by running `make`. These are the same benchmarks used by DFSCQ [10] (a state-of-the art sequential verified file system) and for an unverified NFS server implemented on top of GoJournal [9]. To evaluate the benefit of concurrency, we also evaluate against a "seq txn" variant of DaisyNFS that replaces its per-address locking with a single global transaction lock. In non-concurrent workloads, this variant performs slightly better, demonstrating the overhead of fine-grained locking.

As a baseline, this evaluation uses a Linux NFS server exporting an ext4 file system mounted with `data=journal`

[1]The Dafny implementation of DaisyNFS is at `github.com/mit-pdos/daisy-nfsd`. It imports the transaction system from `github.com/mit-pdos/go-journal`.

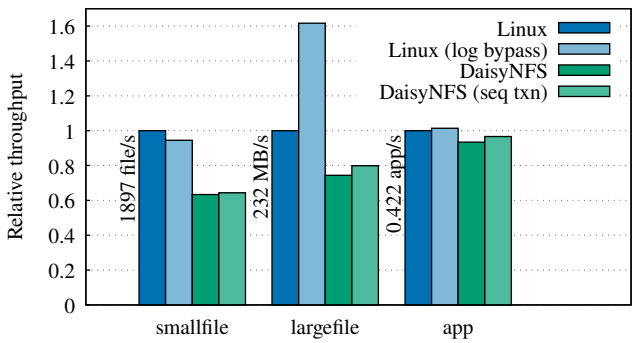


Figure 13: Performance of Linux NFS and DaisyNFS for `smallfile`, `largefile`, and `app` workloads, on an NVMe disk. DaisyNFS achieves comparable performance to ext4 in `data=journal` mode.

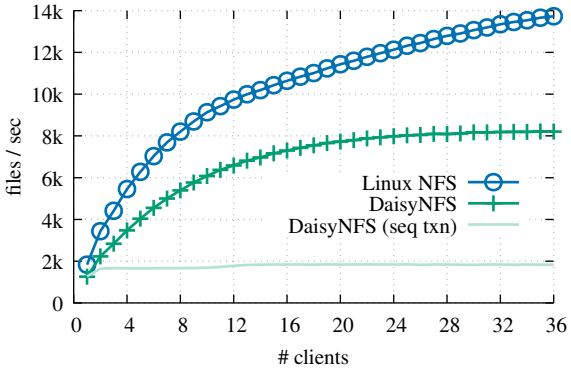


Figure 14: Combined throughput of the smallfile microbenchmark running on an NVMe disk while varying the number of concurrent clients. DaisyNFS's performance scales with the number of cores, though not as well as Linux; both eventually saturate the disk and scale sub-linearly.

mode. The NFS server lets us compare fairly since both go through the Linux NFS client and use the same underlying protocol. Using `data=journal` forces all data to go through the journal and disables log-bypass writes, which ensures that ext4 and DaisyNFS both guarantee NFS RPCs are committed durably when they return. The evaluation also presents results with ext4's log-bypass optimization (in `data=ordered` mode), which gets better performance for some benchmarks but can lose recently written data if the system crashes.

All of these benchmarks were run using Linux 5.15 and Go 1.18.1 on an Amazon EC2 i3.metal instance, which has 72 cores, 512 GB of RAM, and a local 1.9 TB NVMe SSD. To reduce variability we limit the experiment to a single 36-core socket, disable turbo boost, and disable processor sleep states; the coefficient of variation for all experiments is under 5% so we omit error bars for visual clarity.

The results are shown in Figure 13. DaisyNFS gets about 60% the throughput of Linux on the smallfile benchmark, which is intended to be metadata-heavy. The smallfile benchmark repeatedly creates a file, writes 100 bytes to it and syncs the file, then deletes it. Performance is lower than with Linux due to less efficient use of the drive; we used `blktrace` to confirm that Linux issues fewer I/O requests per iteration and that those writes are entirely sequential, unlike with DaisyNFS. Performance is comparable when run on an in-memory disk (not shown in the graph).

DaisyNFS gets comparable throughput to Linux on the largefile benchmark, which is intended to measure bulk data writes. The benchmark creates a 300 MB file by appending repeatedly, then syncs it. Note that in this benchmark ext4 is 60% faster with its log-bypass optimization due to no longer writing all data through the journal. For this workload, the Linux NFS client buffers the entire append process until the final sync, at which point it issues the writes in many chunks in parallel. These RPCs are challenging to support efficiently because they do not arrive at the server in order, so some are past the end of the file. The semantics of such a write are to fill the gap with zeros, but both DaisyNFS and Linux get good performance despite this because they implicitly encode those zeros without even allocating a block.

DaisyNFS achieves good performance on the app workload, which consists of running `git clone` on the xv6 repo followed by `make`. xv6 is an operating system, so building it requires running the usual development tools—`gcc`, `ld`, `ar`—but also running `dd` to generate a kernel image. Builds take about 3s (of which about 1.2s are spent compiling and not in the file system), which are reported as a throughput number so higher is better.

9.2 Scalability

DaisyNFS executes NFS operations concurrently to achieve better performance with multiple cores. The transaction system is built on GoJournal, which already demonstrated scalability. Here we report a similar experiment to demonstrate that DaisyNFS can take advantage of GoJournal's scalability, after accounting for the transaction system's two-phase locking and any overhead added by the transactions themselves. The benchmark used is the smallfile benchmark from §9.1, with a varying number of cores. Because this experiment runs on a physical drive, other threads have a chance to prepare transactions while the journal is committing to disk.

The results are shown in Figure 14. The graph shows that DaisyNFS gets higher throughput with more clients, though its scalability is not as good as the Linux NFS server and its peak throughput is 60% that of Linux. DaisyNFS scales sublinearly due to a lock in GoJournal that serializes installation of writes into disk blocks at commit time. As expected, with a global transaction lock performance does not improve with more clients.

9.3 Testing the trusted code and spec

For the NFS server to satisfy Theorem 1, we trust that (1) the Dafny code is a "safe" use of the transaction system, (2) sequential refinement is correctly encoded into Dafny, (3) the libraries for Go primitives are correctly specified in Dafny, and (4) the unverified Go code calling the Dafny methods

Bug	Why?
XDR decoder for strings can allocate 2^{32} bytes	Unverified
File handle parser panics if wrong length	Unverified
WRITE panics if not enough input bytes	Unverified
Directory REMOVE panics in dynamic type cast	Unverified
Panic on unexpected enum value	Unverified
Concurrent writes can conflict	Unverified
The names `.` and `..` are allowed	Not in RFC 1813
RENAME can create circular directories	Not in RFC 1813
CREATE/MKDIR allow empty name	Specification
Proof assumes caller provides bounded inode	Specification
RENAME allows overwrite where spec does not	Specification

Figure 15: Bugs found by testing at the NFS protocol level.

and implementing the NFS wire protocol is correct. Finally, the user must follow the assumed execution model and run initialization from an empty disk, run recovery after each boot, and the disk should preserve written data and not corrupt it.

Beyond satisfying this formal theorem statement, we want two more things from the implementation and specification: first that the specification as formalized actually reflects the RFC, and second we would like DaisyNFS to be compatible with existing clients, including implementing enough of the RFC's functionality. These fall outside the scope of verification so we cover them with testing.

To evaluate the file system we mounted it using the Linux NFS client and ran the fsstress and fsx-linux tests, two suites used for testing the Linux kernel. In order to look for bugs in crash safety and recovery, we also ran CrashMonkey [30], which found no bugs after running all supported 2-operation tests.

While elsewhere in this paper we interact with DaisyNFS via the Linux client, a collaborator (but not an author) tested it more directly using an NFS-specific testing tool.[2] This testing produces a wider range of requests than are possible via the Linux client. This process helped us find and fix several bugs in the unverified parts of DaisyNFS and in the specification itself. These are reported in Figure 15.

Two of the specification bugs are particularly interesting. The bounded inode bug was due to an `ino` argument of type `Ino`; this type is a Dafny *subset type*, thus adding an implicit precondition that `ino < NUM_INODES`, which is violated by the (unverified) Go code. The fix is to instead use a `uint64` and check the bound in verified code. The RENAME bug was due to having an incomplete specification (and implementation) that did not capture that RENAME should only overwrite when the source and destination are compatible.

9.4 Incremental improvements

DaisyNFS was implemented and verified over the course of three months by one of the authors, until it had support for enough of NFS to run. We added several features incrementally after the initial prototype worked, both to improve performance and to support more functionality. Some of the interesting changes are listed in Figure 16. To improve performance, we switched to operating on the serialized representation of directories directly (decoding fields on demand and encoding in-place) and then added also multi-block directories. We added support for attributes so that the file system stores the mode, uid/gid, and modification timestamp for files and directories. Finally, we implemented the freeing plan described in §7.1.2, which required additional code through the whole stack (but by design no changes to the file-system invariant). We believe additional features such as symbolic links could be added incrementally with modest effort because of sequential reasoning and proof automation.

Feature	Time	Lines
In-place directory updates	2 days	600
Multi-block directories	5 days	800
NFS attributes	4 days	500
Freeing space (§7.1.2)	3 days	1400

Figure 16: Incremental improvements were implemented quickly and without much code (which includes both implementation and proof).

10 Conclusion

This paper presented DaisyNFS, a verified crash-safe, concurrent file system. DaisyNFS was built with verification in mind in two parts: a transaction system called GoTxn, and a file system on top implemented with one transaction per operation. This design allowed us to use the sharpest tool for each part: Perennial for concurrency and crash-safety reasoning and Dafny for sequential reasoning with much proof automation inside a transaction. The specification of the transaction system was designed to support sequential reasoning from Dafny. Overall this approach results in proof overhead of about $2\times$ for the file system part (vs. $20\times$ for the transaction system), allowing us to verify and build a functional file system with good performance.

Acknowledgments

Many people helped improve this paper, including the anonymous reviewers, the PDOS students who gave feedback, Henry Corrigan-Gibbs, and our shepherd, Manos Kapritsos. James Wilcox provided expert debugging assistance. Robert Morris tested DaisyNFS and reported the bugs in Figure 15. This research was supported by NSF awards CNS-1563763 and CCF-1836712.

References

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[2] This framework is part of an unrelated research project so we unfortunately lack space to give details on the methodology itself.

[2] Stefan Bodenmüller, Gerhard Schellhorn, Martin Bitterlich, and Wolfgang Reif. Flashix: Modular verification of a concurrent and crash-safe flash file system. In *Logic, Computation and Rigorous Methods*, pages 239–265. Springer International Publishing, 2021. Festschrift for Egon Börger's 75th Birthday.

[3] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3), May 2007. Festschrift for John C. Reynolds's 70th Birthday.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[5] Tej Chajed. *Verifying a concurrent, crash-safe file system with sequential reasoning*. PhD thesis, Massachusetts Institute of Technology, May 2022.

[6] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–322, Carlsbad, CA, October 2018.

[7] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1037–1051, Phoenix, AZ, June 2019.

[8] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 243–258, Huntsville, Ontario, Canada, October 2019.

[9] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021.

[10] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017.

[11] Dmitri Chkliaev, Jozef Hooman, and Peter van der Stok. Serializability preserving extensions of concurrency control protocols. In *Proceedings of the 3rd International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI)*, pages 180–193, Novosibirsk, Russia, July 1999.

[12] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. `http://pdos.csail.mit.edu/6.828/xv6`.

[13] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 419–434, Carlsbad, CA, October 2018.

[14] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 67–78, Bangalore, India, January 2010.

[15] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, pages 504–528, Maribor, Slovenia, June 2010.

[16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, Savannah, GA, November 2016.

[17] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 646–661, Philadelphia, PA, June 2018.

[18] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 99–115, Banff, Alberta, Canada, November 2020.

[19] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Monterey, CA, October 2015.

[20] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pages 449–465, San Francisco, CA, July 2015.

[21] Dave Hitz, Michael Malcolm, and James Lau. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, San Francisco, CA, January 1994.

[22] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, Santa Clara, CA, February 2015.

[23] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, January 2015.

[24] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[25] Alex Konradi. Performance optimization of the VDFS verified file system. Master's thesis, Massachusetts Institute of Technology, June 2017.

[26] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 348–370, Dakar, Senegal, April–May 2010.

[27] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR)*, page 516–530, Newcastle upon Tyne, UK, September 2012.

[28] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), December 1975.

[29] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent program. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–210, London, United Kingdom, June 2020.

[30] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, October 2018.

[31] Jörg Pfähler. *A Modular Verification Methodology for Caching and Lock-Based Concurrency in File Systems*. PhD thesis, Universität Augsburg, 2018.

[32] David Harver Pollak. Reasoning about two-phase locking concurrency control. Master's thesis, Imperial College London, June 2017.

[33] Gerhard Schellhorn, Gidon Ernst, Jorg Pfähler, Dominik Haneberg, and Wolfgang Reif. Development of a verified flash file system. In *Proceedings of the ABZ Conference*, pages 9–24, Toulouse, France, June 2014.

[34] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–87, Portland, OR, June 2015.

[35] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: Transactional isolation for block storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 23–37, Santa Clara, CA, February 2016.

[36] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016.

[37] The Coq Development Team. *The Coq Proof Assistant, version 8.15*, January 2022. URL `https://doi.org/10.5281/zenodo.5846982`.

[38] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the*

18th ACM SIGPLAN International Conference on Functional Programming (ICFP), pages 377–390, Boston, MA, September 2013.

[39] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helper for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019.

Design and Verification of the Arm Confidential Compute Architecture

Xupeng Li
Columbia University

Xuheng Li
Columbia University

Christoffer Dall
Arm Ltd

Ronghui Gu
Columbia University

Jason Nieh
Columbia University

Yousuf Sait
Arm Ltd

Gareth Stockwell
Arm Ltd

Abstract

The increasing use of sensitive private data in computing is matched by a growing concern regarding data privacy. System software such as hypervisors and operating systems are supposed to protect and isolate applications and their private data, but their large codebases contain many vulnerabilities that can risk data confidentiality and integrity. We introduce Realms, a new abstraction for confidential computing to protect the data confidentiality and integrity of virtual machines. Hardware creates and enforces Realm world, a new physical address space for Realms. Firmware controls the hardware to secure Realms and handles requests from untrusted system software to manage Realms, including creating and running them. Untrusted system software retains control of the dynamic allocation of memory to Realms, but cannot access Realm memory contents, even if run at a higher privileged level. To guarantee the security of Realms, we verified the firmware, introducing novel verification techniques that enable us to prove, for the first time, the security and correctness of concurrent software with hand-over-hand locking and dynamically allocated shared page tables, data races in kernel code running on relaxed memory hardware, integrated C and Arm assembly code calling one another, and untrusted software being in full control of allocating system resources. Realms are included in the Arm Confidential Compute Architecture.

1 Introduction

The use of sensitive private data in many applications from advertising to healthcare, often in the context of machine learning models, has raised concerns regarding the privacy of data in computing. These applications increasingly run on commodity cloud providers. For example, data and computation may be contained in virtual machines (VMs) running on shared hardware in the cloud, relying on a hypervisor to preserve VM isolation to protect applications and their data in VMs.

Software stacks generally require applications to trust system software which they rely on, such as hypervisors and operating systems (OSes). Although hypervisors and OSes are supposed to protect applications and their private data, their large codebases contain vulnerabilities that can risk data confidentiality and integrity. Vulnerable system software running at more privileged levels that can access application data is a significant security issue.

To address this problem, we introduce the *Arm Confidential Compute Architecture (Arm CCA)*. CCA provides *Realms*, secure execution environments that are completely opaque to privileged, untrusted system software such as OSes and hypervisors. CCA retains the ability of existing system software to manage hardware resources for Realms while preventing it from violating Realm confidentiality and integrity. For example, a hypervisor should retain its ability to dynamically allocate memory to or free memory from a Realm VM, but must never be allowed to access the protected memory contents of a Realm VM. CCA guarantees the confidentiality and integrity of Realm code and data in use, that is data in CPU registers and memory, but makes no guarantees regarding their availability. Confidentiality means that any change that a Realm makes to its private data cannot be observed by other Realms or untrusted system software. Integrity means that a Realm will not observe any changes to its private data that it did not make. Because CCA does not guarantee availability, a Realm data access is allowed to halt Realm execution.

CCA avoids hardware complexity by only introducing core hardware mechanisms for attestation and basic address space protection, then relying on firmware to manage the use of those mechanisms. Specifically, CCA introduces *Realm world*, a new physical address space for Realms orthogonal to privilege levels and separate from the existing *Non-Secure (NS) world* used today for running software stacks. Within each world, the normal privilege levels apply and instructions retain their existing semantics, but software in NS world cannot access CPU state and memory used by software in Realm world. CCA introduces a new *Realm Management Monitor (RMM)*, firmware which runs in Realm world at a higher privilege level than Realms. Untrusted system software such as a hypervisor running in NS world can then make requests to RMM to manage Realms, including creating and running Realms. RMM protects the confidentiality and integrity of Realms while handling such requests. System software in NS world is expected to retain full control of the dynamic allocation of hardware resources to Realms, including memory allocation and CPU scheduling.

Because any compromise of RMM could violate the security guarantees of Realms, it is crucial to formally verify its security and functional correctness. However, verifying RMM poses at least four significant challenges. First, RMM employs fine-grained synchronization mechanisms such as hand-over-hand locking to improve performance. Second, RMM has data races and runs on Arm multiprocessor hardware with relaxed

memory behavior. Third, RMM contains both C and Arm assembly code integrated together which call one another freely. Finally, RMM must protect the confidentiality and integrity of Realms even though untrusted system software has full control over the dynamic allocation of Realm resources. Previous verification approaches have not been able to verify system software with these properties [10, 11, 13, 26, 42, 43, 50, 62]

To verify RMM, we introduce VIA (Verification Infrastructure for Armv9-A), which supports four key verification techniques. First, VIA introduces *mover oracle queries* to combine a local CPU model with mover types [44]. These queries encapsulate how operations on other CPUs are interleaved with local CPU operations and can be reordered using mover types to group local CPU operations together. Along with the local CPU model, this allows easier sequential reasoning and modular verification. This makes it possible for the first time to verify hand-over-hand locking with dynamically allocated shared multi-level page tables in system software.

Second, VIA decomposes concurrent code into data race free (DRF) and not-DRF components, then introduces *permutation conditions* for the latter such that proofs on a sequentially consistent memory model will hold on relaxed memory hardware for all concurrent code. Instead of having to verify all of the code directly on relaxed memory hardware, all that is required is to prove that the code satisfies the permutation conditions, which ensure equivalent behavior on sequentially consistent and relaxed memory hardware. VIA allows any permutation conditions to be defined, supporting verification of a broad class of programs.

Third, VIA bridges incompatibilities between C and assembly code due to CPU register state being hidden by the former but explicitly used by the latter. To accomplish this without dependencies on a specific compiler, VIA introduces a register accounting mechanism to correctly verify integrated C and Arm assembly code. It leverages the machine-level procedure call standard for the Arm instruction set to specify how registers are potentially used when assembly code calls a C function or is called by a C function. VIA tracks CPU register state across invocations of both C and assembly code primitives, capturing any information flow through CPU registers even if hidden by C semantics.

Finally, VIA introduces an ideal/real paradigm for verifying security properties that can be applied to Realms, even though untrusted system software is in full control of system resources and can reclaim system resources such as memory without Realm permission, breaking noninterference. VIA defines an *idealized secure machine model* that supports declassification. Realm private data is stored in physically isolated memory and CPU registers. Data channels, governed by security policies, are used to exchange information between Realms and untrusted software. We can then prove the security guarantees of Realms by verifying that the implementation refines its specification and the real system captured by the specification simulates the idealized secure machine model. This approach allows us to prove, for the first time, the integrity and confidentiality of Realms. A key feature of the proof is that it only needs to trust the specification of the small idealized secure machine model; the much larger specification of the real system does not need to be trusted.

We implemented, evaluated, and verified an early prototype of CCA firmware. Although CCA hardware is not yet available, we demonstrated CCA on a functionally accurate Arm Fast Model with CCA support. We modified the Linux KVM hypervisor [19–21] to run on CCA and manage Realm VMs, and ran various VM workloads on the model. We also ported CCA firmware to current Arm hardware to obtain preliminary data on CCA performance, which shows that KVM on CCA incurs modest overhead versus vanilla KVM on real application workloads. We verified the correctness of both the C and Arm assembly CCA firmware implementation, including RMM, proving its implementation refines its specification through 43 abstraction layers. We then proved the specification is equivalent to the behavior of the idealized secure machine model to verify the confidentiality and integrity guarantees of Realms. The proof only needs to trust roughly 200 lines of Coq specification, making the formal security guarantees easy to read and understand. This is the first proof of the security guarantees of a confidential computing architecture. Realms will be included in Armv9-A, the next version of the Arm architecture.

2 Threat Model

We consider an attacker without physical access to the machine and assume the attacker's goal is to compromise the confidentiality and integrity of VM data. Confidentiality and integrity attacks in scope include compromising the hypervisor or any other software to read or modify private VM memory or register state, including by controlling DMA-capable devices, or via memory remapping and aliasing attacks. We assume a VM does not voluntarily reveal its own private data whether on purpose or by accident, but attacks from other compromised VMs, including confidentiality and integrity attacks, are in scope. Availability attacks by a compromised hypervisor are out of scope. Protection against known software error injection attacks and side-channel attacks require appropriate usage of architectural mitigations and are beyond the scope of this paper. DRAM attacks, such as cold boot attacks, live probing, or replay, require additional hardware and are outside of the scope of the threat model.

3 CCA Design

A key challenge with introducing Realms is how to provide backwards compatibility with a widely-used existing architecture that, like other CPU architectures, was designed based on the fundamental assumption that more privileged levels have greater control and access than less privileged levels of

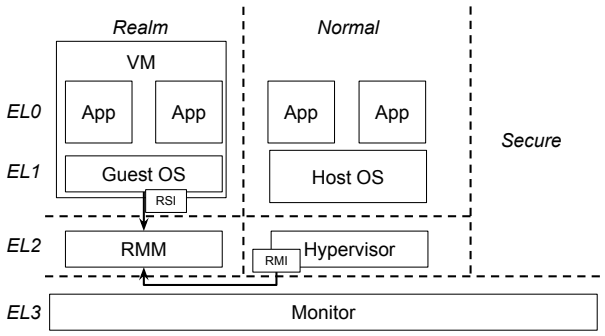


Figure 1: Arm Confidential Compute Architecture.

software. One issue is understanding the potential interactions of Realms with all the features in the Arm architecture. For example, debug registers defined in the Arm architecture are explicitly designed to allow hypervisors to peer into VM state, which is fundamentally at odds with Realms. The behavior of each instruction could be redefined in the context of Realms, but this would be an enormous undertaking with unclear compatibility implications, given that the Arm instruction set was designed over multiple decades.

Another issue is how to provide memory protection and isolation for Realms. The way this works for VMs is that hypervisors manage *nested page tables (NPTs)* [9] to isolate physical memory between VMs and protect hypervisor memory from VMs. The physical addresses perceived by a VM are *intermediate physical addresses (IPAs)*, which are translated by an NPT to physical addresses for the hardware. Physical memory not mapped to the NPT is not accessible to the VM. However, NPTs are under full control of the untrusted hypervisor, providing no protection against hypervisor access to VM data. While it would be possible to introduce an additional data structure to track memory ownership for each frame of physical memory [3], this approach comes with several problems. First, the amount of information required for each frame of memory would be substantial and significantly impact TLB design and performance. Second, this data structure would have to be managed either via a separate more privileged software entity than the hypervisor or via complex instructions capable of capturing measurements of data assigned to a Realm. Such complex CISC-like instructions would almost certainly require introducing extensive microcode into an architecture, which does not currently use any.

CCA avoids these problems by only introducing simple hardware mechanisms orthogonal to existing privilege levels and then relies on firmware to manage the use of those mechanisms. This reduces hardware complexity at the cost of depending on the firmware for the security guarantees of the architecture. As a result, verifying CCA firmware is of crucial importance.

Figure 1 shows how CCA extends the Arm architecture. Armv8-A provided two statically partitioned worlds, NS world used by most software stacks and Secure world to host

Security State	PAS			
	NS	Secure	Realm	Root
NS	Allow	Block	Block	Block
Secure	Allow	Allow	Block	Block
Realm	Allow	Block	Allow	Block
Root	Allow	Allow	Allow	Allow

Table 1: CCA access control policy. The entity accessing a granule belongs to a security state, while the PAS is a property only of the granule being accessed.

platform security services [4]. CCA introduces Realm world, which is fully compatible with NS world so that existing software stacks that run in NS world can also run in Realm world. CCA provides three privilege levels in each of the NS, Realm and Secure worlds: EL0 for user, EL1 for kernel, and EL2 for hypervisor. Because Realm and Secure worlds are mutually distrusting, CCA introduces a fourth, more privileged Root world to manage switching between the other worlds.

Each world has its own *Physical Address Space (PAS)*. Each 4 KB frame of physical memory, which we refer to as a *memory granule*, belongs to one PAS at any given time. Individual memory granules can be dynamically transitioned from NS PAS to Realm PAS; there is no static partitioning of resources between NS and Realm worlds. Hardware performs a PAS check on each memory access against a *Granule Protection Table (GPT)* that tracks the PAS of each memory granule and enforces the access control policy shown in Table 1, forbidding invalid accesses. NS world can only access its own memory. Realm and Secure worlds can access their own respective memory and NS memory, but cannot access each other's memory. CCA hardware requires all DMA accesses be subject to GPT checks, protecting the Realm PAS against DMA-based attacks. We focus on the interactions between NS and Realm worlds and omit further discussion of Secure world due to space constraints.

CCA relies on two trusted firmware components: RMM and the *EL3 Monitor (EL3M)*. RMM runs at EL2 in Realm world. It controls the execution of Realms and provides services to untrusted system software running in NS world. It isolates Realms from each other using existing virtualization technologies such as NPTs and CPU register save/restore sequences. Because RMM only enforces the security guarantees of CCA, it can be orders of magnitude smaller than bare-metal hypervisors which must also provide virtualization functionality. For example, to run Realm VMs, RMM protects the confidentiality and integrity of Realms while relying on existing hypervisors for everything else, including resource allocation and scheduling, physical hardware support, and complex device emulation.

EL3M runs in Root world at EL3, the highest level of privilege. It is responsible for context switching CPU execution among the three other worlds and managing the GPT. EL3M can access memory in any PAS. Only EL3M can change the PAS of a granule, which involves updating its entry in the GPT. Software running in the three other worlds can issue a *Secure Monitor Call (SMC)* to EL3M to request a PAS change.

In the current version of CCA, the Realm isolation boundary

Command	Description
`Version`	Query RMI ABI version.
`Granule.Delegate`	Change granule (from NS) to Delegated.
`Granule.Undelegate`	Change granule (from Delegated) to NS.
`Realm.Create`	Create Realm Descriptor (RD).
`Realm.Destroy`	Destroy Realm identified by RD.
`Realm.Activate`	Change Realm (from New) to Active.
`REC.Create`	Create Realm Execution Context (REC).
`REC.Destroy`	Destroy REC.
`REC.Run`	Enter REC (i.e. run VCPU).
`Data.CreateUnknown`	Change granule to Data with unknown content.
`Data.Create`	Change granule to Data, copy NS content.
`Data.Destroy`	Change Data granule to Delegated, zeroed.
`RTT.Create`	Create Realm Translation Table (RTT).
`RTT.Destroy`	Destroy RTT.
`RTT.MapProtected`	Map Data granule in RTT.
`RTT.UnmapProtected`	Remove mapping from RTT.
`RTT.MapUnprotected`	Map NS granule in RTT.
`RTT.UnmapUnprotected`	Remove NS mapping from RTT.
`RTT.ReadEntry`	Return content of an RTT entry.

Table 2: RMM Realm Management Interface (RMI).

is at the level of entire VMs; applying Realms to secure other entities such as containers [59] is future work. Similar to normal VMs, a Realm VM can concurrently run multiple virtual CPUs (VCPUs) and the number of Realm VMs on a system is only limited by the amount of physical memory available, not by any arbitrary limits. The untrusted hypervisor always has the ability to stop scheduling a Realm and can always reclaim memory assigned to a Realm, but in no circumstances does it have access to Realm CPU or memory state.

This split of responsibility between an untrusted hypervisor and RMM, where the untrusted hypervisor allocates memory, and RMM provides integrity and confidentiality guarantees for the data and code stored in that memory, is accomplished through a simple but powerful delegation concept. The hypervisor *delegates* memory to Realm world, and *undelegates* memory back to NS world. All memory used by Realms must first be delegated by the hypervisor; RMM does not itself manage a pool of memory for Realms. Once memory is delegated to Realm world, the hypervisor can request RMM to use it for various purposes, such as storing metadata or data for a Realm. Whenever a memory granule is delegated to Realm world but not used by RMM, RMM ensures that the granule contains only zeros, reducing the risk of accidental information flow when a granule is reused or undelegated.

RMM provides a *Realm Management Interface (RMI)* for the hypervisor to request RMM to delegate memory, create Realms, execute Realms, and allocate memory to Realms. Each RMI command is implemented as an SMC, so when the hypervisor invokes the command, it traps to EL3M, which in turn switches execution to RMM in Realm world to handle the command. Upon completion of the RMI command, RMM issues an SMC to EL3M, which switches execution back to the hypervisor in NS world. Table 2 lists the RMI commands.

RMM must know the state of each memory granule on the system to uphold the security guarantees of Realms, which it accomplishes by maintaining its own *Granule Status Table (GST)* to track the delegation status and current use of each granule. RMM uses the GST to ensure that a granule is in a valid state to perform the requested action. For example, when the hypervisor delegates a memory granule, RMM checks its GST to confirm the granule has not already been delegated, then issues an SMC to EL3M to request a change to Realm PAS. EL3M checks that the granule is currently in NS PAS, then updates the GPT to move it to Realm PAS. Finally, RMM updates its GST to record that the granule has been delegated. If the hypervisor attempts to delegate a granule which is already delegated, or undelegate a granule which is in active use by RMM, RMM returns an error code to the untrusted hypervisor. This pattern of checking valid states and either performing a discrete action or returning an error is used for all RMI commands, allowing RMM to remain in overall control of the consistency of the system, while complex logic for policy and resource allocation remains in the hypervisor. Unlike the GPT, the GST is not checked by hardware and is only a software bookkeeping mechanism. By maintaining a separate GST from the GPT, the GPT can be kept simple so that it only needs to contain information required for hardware-enforced checks.

The hypervisor creates Realms, *Realm Execution Contexts (RECs)*, and *Realm Translation Tables (RTTs)* using the respective commands in Table 2. RECs correspond to VCPUs and RTTs correspond to NPTs for normal VMs. RTTs are Arm stage 2 page tables that translate from an IPA to a physical address. RTTs use the same format and topological layout in Realm world as NS stage 2 page tables, but also provide a bit which allows Realms to access NS granules under the control of RMM, for example, for virtual I/O between a Realm and the hypervisor. On each of the Realm, REC, and RTT create commands, RMM checks the GST entry for the address provided to confirm the granule is already delegated, and updates the GST entry to track that it is being used for Realm, REC, and RTT metadata, respectively. We refer to a Realm's metadata as its *Realm Descriptor (RD)*.

A Realm provides a *Protected Address Range (PAR)* within its IPA space, which RMM ensures can only be mapped to Realm PAS granules. For accesses within the PAR, RMM guarantees confidentiality and integrity to the Realm; outside the PAR, the hypervisor is free to map NS PAS granules or emulate accesses. This provides an OS running inside a Realm VM with a reliable mechanism to determine whether it is accessing its own private memory, or memory which can be shared with untrusted agents, for example, buffers used for untrusted DMA with virtual or physical network and block devices.

During Realm creation, the hypervisor can assign a granule to the Realm at a specific IPA and copy data to it from an NS granule. The IPA and data are cryptographically hashed and the hash is included in the attestation token of the Realm. The attestation token allows a Realm owner to reason about its initial state and content. Once a Realm has been activated, the measurement is fixed, and memory can only be added to otherwise unused IPAs with unknown content. We refer

to delegated granules used to store data for a Realm as *Data granules*. The hypervisor can request that RMM maps NS granules outside the PAR at any time. Physically contiguous delegated memory can be mapped to a Realm in blocks larger than 4 KB granules to optimize TLB usage.

The hypervisor can reclaim memory from a Realm at any time. RMM zeros a granule before undelegating it and returning it to the hypervisor. Subsequent accesses from a Realm to the IPA where the memory was reclaimed result in a stage 2 abort to RMM which prevents further execution of the Realm and preserves the CCA integrity guarantee. The hypervisor cannot subsequently map a granule to a previously-backed IPA within a PAR without Realm permission.

As a system designed to scale to many cores, RMM makes extensive use of fine-grained locking to support a high degree of concurrent operation. For example, each memory granule has its own lock so many granule operations can be done in parallel. Similarly, an RTT is a multi-level page table, for which each level has its own lock, and hand-over-hand locking is used to support concurrent operations on RTTs, as discussed in Section 4.1. For example, two Realm VCPUs can each cause a stage 2 page fault at the same time but at different IPAs, which can be resolved by the hypervisor in parallel on two CPUs to improve performance. This is a key requirement to support large Realms. Although most of RMM is written in C, Arm assembly code is also used to implement memory accesses with acquire/release semantics where lockless concurrent accesses are used for performance reasons, and to implement the locking primitives themselves.

CCA firmware is designed for security following best practices. Systems such as Linux map all physical memory to the kernel page table. RMM and EL3M do not. RMM's own page table statically maps code and metadata exclusively accessed by RMM, such as the GST and locks for each granule. Additional entries in RMM's page table are used to statically assign a virtual address range to each physical CPU in the system, resulting in a fixed number of virtual address slots per CPU. Memory is then mapped on demand when needed. RMM maps Data granules and metadata granules, such as RD and REC, on demand, and unmaps them once the respective operation is completed. EL3M's own page table only statically maps the EL3M code, a small fixed size stack, and the GPT; no other memory is mapped to its page table. Furthermore, SMC parameters are only interpreted as values in EL3M, never as pointers used to access memory. Even if a bug is introduced in some future version of CCA firmware that is not completely verified, these defense-in-depth measures make it much harder for a return-oriented or jump-oriented programming attack to succeed.

4 VIA Framework

Because CCA relies on firmware to guarantee the security of Realms, we verify that firmware, namely RMM and EL3M. We prove the CCA firmware implementation refines its layered specification in Coq, then use the top-level specification to prove the system's security properties hold for the implementation. To accomplish this, we developed the VIA verification framework, which supports layered verification of CCA firmware. VIA introduces four key verification techniques: mover oracle queries, relaxed memory support via permutation conditions, register accounting for C and assembly code integration, and a new ideal secure system model for proving security properties that cannot be verified using traditional noninterference-based approaches.

4.1 Mover Oracle Queries

To verify RMM, it is essential to simplify reasoning about possible interleavings of executions of concurrent software across multiple CPUs. For example, RMM uses hand-over-hand locking to synchronize access to RTTs, which are 4-level page tables, allowing multiple CPUs to manipulate the same page table concurrently. Figure 2 shows the steps to allocate delegated granules as new level `T1`, `T2`, and `T3` tables of a Realm's RTT using `RTT.Create` and then, in step 4, allocate a delegated granule to the Realm for its data and map its physical address to the leaf-level `T3` table using `RTT.MapProtected`, which would typically occur on a page fault. Figure 2 also shows how step 4 uses hand-over-hand locking, in which RMM first acquires `T0`'s lock so it can lookup and acquire `T1`'s lock and release `T0`'s lock. It can then lookup and acquire `T2`'s lock and release `T1`'s lock, so it can lookup and acquire `T3`'s lock and release `T2`'s lock, and finally update `T3`'s page entry. At the same time, RMM running on other CPUs can do other page table operations, such as acquiring `T0`'s lock to work on a different level 1 table.

To verify the page table operations with hand-over-hand locking, we need to reason about the correctness of all possible interleavings of operations. However, reasoning about all possible interleavings of all operations all at once is too difficult to do for a system as complex as RMM. To address this problem, VIA introduces mover oracle queries, a new mechanism that combines the power of local CPU reasoning with mover types [44], building on previous work on CertiKOS [24–27] and CSPEC [10].

To explain how mover oracle queries work, consider first an explicit multiprocessor machine model, whose machine state consists of per-physical CPU private state (e.g., CPU registers) and a global logical log, a serial list of events generated by all CPUs throughout their execution. Instead of explicitly modeling shared objects, events incrementally convey interactions with shared objects, whose state may be calculated by replaying the logical log. An event is emitted by a CPU and appended to the log whenever that CPU invokes a primitive that interacts with a shared object. Our abstract machine is formalized as a transition system, where each step models some atomic computation taking place on a single CPU; concurrency is realized by the nondeterministic interleaving of steps across all CPUs. However, reasoning

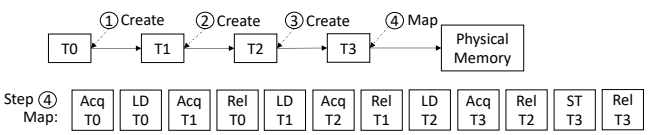


Figure 2: Page table creation and hand-over-hand locking execution.

about interleavings directly with multiple CPUs is difficult.

To simplify reasoning about all possible interleavings, we instead lift multiprocessor execution to a local CPU model, which distinguishes execution taking place on a particular CPU from its concurrent environment [27, 36, 42]. All effects coming from the environment are encapsulated by and conveyed through an *event oracle*, which yields events emitted by other CPUs when queried. Querying the event oracle can be thought of in the context of the explicit multiprocessor machine model as returning events from the global log generated by all other CPUs; only new events since the last query are returned. How the event oracle synchronizes these events is left abstract, its behavior constrained only by rely-guarantee conditions [35]. Since the interleaving of events is left abstract, our proofs do not rely on any particular interleaving of events and therefore hold for all possible concurrent interleavings.

A CPU captures the effects of its concurrent environment by querying the event oracle between local CPU steps. A CPU only needs to query the event oracle when interacting with shared objects, since its private state is not affected by these events. In other words, the CPU repeatedly performs two steps when interacting with shared objects: querying the event oracle to obtain events from other CPUs, then generating a local CPU event. The result is a composite log of events from other CPUs interleaved with events from the local CPU. This is equivalent to the logical log in the explicit multiprocessor model, but without the complexity of directly reasoning about multiple CPUs.

If possible, we would like to move the interleaved event oracle queries out of the way of the local CPU events so we can use sequential reasoning regarding the local execution of any given CPU. By using mover types, we can identify how we can reorder event oracle queries with respect to local CPU events without changing the machine's behavior. Thus, these queries are mover oracle queries. We classify all local CPU events in the composite log as RightMover, LeftMover, or NoneMover. Mover oracle queries can be reordered before a RightMover and after a LeftMover. For example, acquiring a lock is a RightMover because if other CPUs do something after acquiring the lock on the local CPU, they must be able to do the same thing before acquiring the lock. The oracle queries which capture the other CPUs' events can be reordered before acquiring the lock. Mover oracle queries cannot be reordered with a NoneMover. For example, an oracle query followed by a NoneMover then a LeftMover cannot be reordered after the LeftMover.

VIA can then reduce the interleaving of events in the log that need to be considered in two ways, which we refer to as *log refinement*. First, we can reorder oracle queries with local CPU

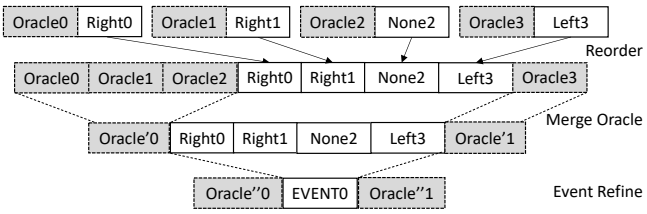


Figure 3: Log refinement with mover oracle queries.

events based on the local events' mover types. By reordering, consecutive oracle queries will be merged to one. Second, we can prove local sequences of events generated by the machine refine an aggregate local event generated by a higher-level machine. This refinement can be applied to any arbitrary CPU, therefore, it applies to all CPUs, so that the entire log of events refines the log of the higher-level aggregate events.

Figure 3 shows an example of log refinement to reduce interleavings of events across CPUs into an atomic event. We identify the mover type of each local event, i.e. [`Right 0`, `Right 1`, `None 2`, `Left 3`], and initially query the oracle before each event. Based on the mover types, we can reorder all oracle queries before the NoneMover to the beginning, and all remaining queries to the end, such that the log before and after reordering have the same machine behavior. We then define a new oracle that can be queried to return the consecutive events from the previous oracle queries [`Oracle 0`, `Oracle 1`, `Oracle 2`], allowing those events to be merged into a single oracle query [`Oracle' 0`]. We then refine the local sequence of events [`Right 0`, `Right 1`, `None 2`, `Left 3`] into a single higher-level aggregate local event `EVENT 0`. This can be done for all CPUs so we can reason further only using the higher-level aggregate event `EVENT 0` with oracle queries `Oracle" 0` and `Oracle" 1` that also return higher-level aggregate events, instead of the many `Left/Right/None` events of lower-level machine.

4.2 Permutation Conditions

To verify RMM, we must account for the relaxed memory behavior of the Arm architecture on code that is not data race free (DRF). For example, Figure 4 shows how a Realm's list of `RECs` is updated in `REC.Create`, `REC.Destroy`, and `Realm.Destroy` without holding a common lock. Each Realm's `RD` has a `RECLIST` (`rd->rec_list`), an array that stores the pointers to all its `RECs`. The `RECLIST` can be referenced from both the Realm's `RD` and each of the Realm's `RECs` (`rec->rec_list`). Each `REC` records its index in the `RECLIST` (`rec->id`). `RD`'s `counter` keeps tracking of how many `RECs` are in a Realm. The hypervisor must destroy all `RECs` of a Realm before destroying its `RD` because once `RD` is destroyed, the Realm can no longer be referenced. Access to the `RECLIST` is not synchronized by its own lock, to avoid potential deadlock issues due to needing to hold multiple locks. Instead, in `REC.Create`, the `RD`'s lock must be held to insert a new `REC` in `RECLIST` to ensure mutual exclusion. However, in `REC.Destroy`, the `REC`'s lock is held instead of the `RD`'s

```
Rec.Create(rd, id) {
    acq(rd->lock)
    ...
(a) if (rd->rec_list[id] == NULL) {
(b)   rd->rec_list[id] = NEW_REC;
(c)   atomic_inc(rd->counter);
    ...
    rel(rd->lock);
}
```

```
Rec.Destroy(rec) {
    acq(rec->lock);
    ...
(d) rec->rec_list[rec->id] = NULL;
(e) atomic_dec(rec->rd->counter);
    rel(rec->lock);
}
```

```
Realm.Destroy(rd) {
    acq(rd->lock);
    ...
(f) if (rd->counter == 0) {
      // rec_list should be EMPTY
(g)   destroy(rd->rec_list);
    ...
    rel(rd->lock);
}
```

Figure 4: Pseudo code of `RECLIST` data races, marked in bold blue.

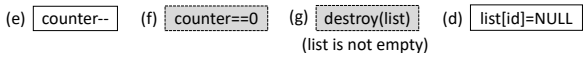
locks when clearing the `REC`'s entry from the `RECLIST` so that multiple CPUs can destroy different `REC`s of the same Realm concurrently. Furthermore, the `RD`'s `counter` is increased or checked in `REC.Create` and `Realm.Destroy` while holding `RD`'s lock, but it is decreased in `REC.Destroy` without holding any lock. As a result, data races can occur when concurrently executing `REC.Destroy` with `REC.Create` or `Realm.Destroy`.

To address this problem, VIA builds on VRM [57]. VRM verifies programs on Arm relaxed memory hardware that are DRF except for synchronization methods and virtual memory hardware. VRM verifies a program on a sequentially consistent (SC) multiprocessor hardware model, defines and proves that a fixed set of conditions hold for the program running on relaxed memory hardware, and proves that the conditions guarantee that the program has the same behavior on SC and relaxed memory hardware so that its SC proofs also hold for relaxed memory hardware.

VIA generalizes this approach for programs that are not DRF. It ensures that such a program will have the same behavior on SC and relaxed memory hardware by first decomposing the program into components that are DRF and not DRF. Previous work already shows that the DRF components will have the same behavior on SC and relaxed memory hardware [57]. VIA then introduces *permutation conditions* $\mathcal{P}$ on the non-DRF components such that $\mathcal{P}$ can be verified to hold for the program on relaxed memory hardware, and $\mathcal{P}$ can be proven to guarantee that the non-DRF components will have the same behavior on SC and relaxed memory hardware. Our experience suggests that even for programs that are not DRF, only a small percentage of the code in these programs is not DRF, so non-DRF programs can be verified on relaxed memory hardware by only proving a small number of permutation conditions in practice. This observation holds for RMM, in which almost all of the code is DRF.

VIA uses VRM's extended Promising Arm model [57] to model Arm's relaxed memory hardware, such that $\mathcal{P}$ needs to be verified against all instruction permutations of the program allowed by VRM's Promising Arm model. Unlike VRM which defines a fixed set of conditions that do not all hold for RMM, VIA allows any condition $\mathcal{P}$ to be specified for non-DRF components that will result in their behavior being in the same on SC and relaxed memory hardware and that can be proven to hold for the program on relaxed memory hardware. The condition is essentially a constraint based on the program's semantics that restricts the possible instruction reorderings that can occur on relaxed memory hardware so that resulting program behavior is the same on SC and relaxed memory hardware.

For example, to handle the non-DRF code in Figure 4, we identify $\mathcal{P}$ to be when `Realm.Destroy` finds `rd->counter` equals 0, `rd->rec_list` must be empty. This is necessary because `rd->rec_list` must be empty when destroying it in `(g)`, otherwise the system may crash due to reclaiming non-empty memory. Since `REC.Create` and `Realm.Destroy` use the same lock, data races can only occur when either runs concurrently with `REC.Destroy`. We prove each function always behaves the same on SC and relaxed memory. For `REC.Create`, since `(b)` and `(c)` cannot be reordered with `(a)` due to the branch dependency, as required by Promising Arm, its possible executions are `(a)(b)(c)` or `(a)(c)(b)`. Since `(a)` confirms that `rec_list[id]` is empty, all concurrent `REC.Destroy` on other CPUs must destroy slots other than `id` because `REC.Destroy` will only work if the `rec` exists, which must be a non-empty slot in the `rec_list`. Therefore, swapping `(b)` and `(c)` will never change any CPU's behavior and `(a)(c)(b)` is equivalent to `(a)(b)(c)`, which is the order on SC. For `REC.Destroy`, if `(e)` executes before `(d)`, $\mathcal{P}$ will be broken because when `Realm.Destroy` checks `counter` concurrently on other CPUs, it may find `counter` is 0 but `rec_list` is not empty, as shown below:



This was actually a real bug in the prototype implementation of RMM. Therefore, we must enforce that `(d)` always executes before `(e)` by adding a barrier between them so it must follow program order as on SC. For `Realm.Destroy`, the proof is trivial because the branch dependency between `(f)` and `(g)` guarantees that they execute in program order as on SC. Therefore, this non-DRF code will not generate more behavior on relaxed memory hardware than on SC.

4.3 Register Accounting

To verify CCA firmware with both C and assembly code, we must account for the interactions of C and assembly code primitives that call one another across language boundaries. However, C code hides the details of how it uses CPU registers, as the use of registers during C code execution is decided by the implementation of specific C compiler used. Although register behavior is not expressed by C language semantics, ignoring it causes problems when attempting to verify programs in which C and assembly code call one another, as shown in Figure 5, which illustrates a real bug in the original prototype RMM implementation detected during our verification. Existing verification approaches cannot support bidirectional calls between C and assembly code, such that the example in Figure 5 would be erroneously verified without detecting the information leakage [10, 11, 23, 26, 37, 42, 43, 46].

To address this problem, VIA introduces a novel register accounting mechanism to correctly verify integrated C and Arm assembly code while making minimal assumptions

```
ENTRY(store_inner):
    str x5, [x1]
    ret
ENDPROC(store_inner)
```
Spec: mem[%x1] = %x5;

```
void store_c() {
  int s = secret;
  // s stored in x5
  store_inner();
}
```
Spec: mem[%x1] = %x5;

```
ENTRY(store_outer):
    mov x5, #0
    bl store_c
    ret
ENDPROC(store_outer)
```
WRONG Spec: mem[%x1] = 0;

Figure 5: An example of incorrectly combining C and assembly specifications. Assembly function `store_outer` clears register `x5` to 0, then calls C function `store_c`. `store_c` calls assembly function `store_inner`, which stores register `x5` into memory. The intended behavior is that the value 0 will be stored to memory. The actual behavior is that `x5` stores C temporary variable `s` which contains secret data, resulting in undetected information leakage.

regarding compiler behavior. VIA leverages the *Arm64 Procedure Call Standard (AAPCS64)* [7] to specify how registers are potentially used when assembly code calls a C function or is called by a C function. It then conservatively marks all registers used by C code whose values cannot be determined based on AAPCS64 as of `Unknown` value, and requires assembly code to not depend on registers with `Unknown` values.

AAPCS64 constrains how some Arm registers are used. In CCA firmware, C functions pass no more than eight integer or pointer parameters and return an integer or pointer. For such functions, AAPCS64 specifies that a C compiler will only pass parameters through registers `r0-r7` and save the return value in `r0`. It also specifies registers that must have their values preserved through a function call, namely all callee-saved registers `r19-r29` and the stack register `sp`. The use of other general-purpose registers (GPRs) may depend on the specific C compiler implementation.

For an assembly function that calls a C function, VIA checks that the assembly code does not read any `Unknown` registers. Legal assembly code can either keep such `Unknown` registers untouched or overwrite them before using them. VIA uses AAPCS64 to model the register behavior of the C function by identifying register `r0` as containing the return value, and registers `r19-r29` and `sp` as preserving the values. It marks the values of other registers after the C function call as `Unknown`, including caller-saved registers `r1-r18` and the link register `lr`.

For an assembly function that can be called from a C function, VIA checks that its behavior does not depend on `Unknown` registers, and that it obeys AAPCS64 C calling conventions so that it will not cause unexpected behavior in its caller. VIA checks that (1) callee-saved registers `r19-r29` and `sp` preserve the values; (2) the program counter `pc` after the call is equal to `lr` before the call so the assembly primitive returns like a function call; (3) if the caller expects a return value, `r0`'s value is never `Unknown`; and (4) the assembly code behavior remains the same if we initialize all GPRs to `Unknown` except for those carrying parameters. The last condition implies that the assembly code does not read any `Unknown` registers, except for saving and restoring callee-saved registers.

VIA also supports GNU Compiler Collection (GCC) inline assembly extensions within a C function. This is used in inline assembly memory accessors in RMM which guarantee

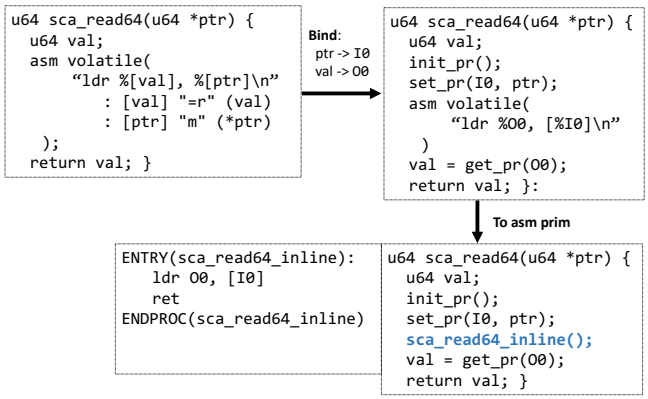


Figure 6: Translation of parameterized inline assembly.

atomicity or memory order semantics, as shown in the `sca_read64` example in Figure 6. `sca_read64` implements a 64-bit single-copy-atomic read in one line of assembly code plus an interface, which can specify a list of input registers, output registers and clobbered registers. VIA translates inline assembly code into an assembly function according to the interface constraints; "r", "Q", and "m" constraints are currently supported. It then checks its correctness like any other assembly function.

Translation is done using a set of logical registers `I0-In` for inputs and `O0-On` for outputs so that verification does not depend on the specifics of GCC register assignment. Input registers are defined read only. VIA also defines abstract accessors `init_pr`, which initializes all logical registers to `UNKNOWN`, `set_pr`, which writes to a register, and `get_pr`, which reads from a register. As shown in Figure 6, the translated `sca_read64` function first calls `init_pr` for initialization, saves parameters to input registers by calling `set_pr`, uses the input and output registers in the assembly code, and gets the return value from the output register by calling `set_pr`.

For simplicity, VIA imposes additional requirements to guarantee GCC generates correct machine code whose behavior is the same as VIA's translated code. VIA forbids inline assembly code from explicitly using any GPRs or goto labels. For inline assembly with multiple instructions, VIA enforces that all output registers are constrained by "&" or "+". Thus, an output-only register never doubles as an input register, and the same register is used for input and output of an operand. This avoids any unexpected overlap in the assignment of input and output registers [53].

Finally, because assembly code functions may be at the interface to outside programs that are untrusted, VIA enforces that all register values are not `Unknown` when returning from those assembly functions. This ensures that there is no unintentional information leakage from assembly code functions to untrusted programs through registers with `Unknown` values.

4.4 Ideal Secure System Model

CCA protects the confidentiality and integrity of Realms' private data during their lifetime. Confidentiality means any

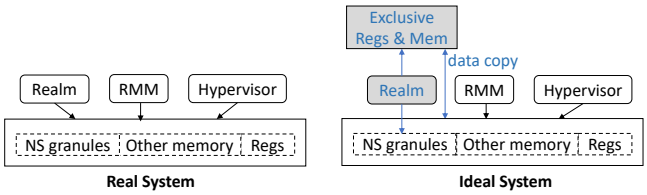


Figure 7: The real and ideal secure system model.

change a Realm makes to its private data is only observable by that Realm. Integrity means a Realm will not observe any changes to its private data that it did not make, but does not imply availability; data access should either fail or return the data previously stored. The confidentiality definition is standard, but the integrity definition allows untrusted software to modify a Realm's private data as long as the Realm does not observe the change. For example, to reclaim memory from Realms, a hypervisor can unmap a Realm's private data without the Realm's permission. This is allowed because the Realm's access to the unmapped data will trigger a page fault so the Realm cannot observe future changes to the data content. However, this breaks noninterference, which therefore cannot be used to to prove security as is done for other verified systems [16, 23, 29, 34, 42, 49, 55].

To address this problem, VIA introduces an ideal/real paradigm, shown in Figure 7, inspired by the idea from formal verification of separation kernels [22, 30]. The *real system* is defined by the RMM top-layer specification, which builds on and incorporates EL3M, in which all memory and CPU registers are shared by Realms, RMM, and the hypervisor. The *ideal system* is defined by an ideal system model specification, in which each Realm has its own exclusive memory, and each REC of the Realm has its own exclusive CPU registers, while other software can only access the same non-exclusive memory and registers as in the real system.

If each Realm only accesses its exclusive memory and registers in the ideal system, we could then show that RMM guarantees confidentiality and integrity by proving that the real system simulates the ideal system. This would mean that each Realm only accesses its exclusive memory and registers in the real system as well, so nothing other than a Realm can access its own data. However, such a simplistic model does not work in practice. For CCA, we need a model that allows declassification so Realms can access NS granules for initialization and I/O, and CPU registers can be used to pass parameters between Realms and RMM, or Realms and the hypervisor.

VIA introduces a new ideal system model for Armv9-A that supports declassification of memory and registers based on a set of well-designed rules that define when declassification is allowed. The model has six declassification rules, listed in Table 3. In this model, Realm exclusive memory consists of all memory in its PAR and exclusive CPU registers consists of all registers accessible by a Realm or that can affect its execution, such as system registers. A Realm will only access its exclusive memory and registers, unless it accesses a granule outside its PAR or it accesses a granule or register that is `Unknown`. If it accesses memory outside its PAR, the Realm will access non-exclusive memory directly. If it accesses a granule or register that is `Unknown`, the data will be copied from a special initialization buffer or non-exclusive register, respectively, before accessing it. A granule is `Unknown` if it is not yet initialized. A register is `Unknown` if it is used by the Realm to communicate with RMM or the hypervisor. For example, when a Realm invokes a hypercall, it exposes the arguments in registers `r0-r6`, which RMM will provide to the hypervisor, then return the results back in those registers. Marking a granule or register as `Unknown` is used to represent declassification in the model.

Type	Rule
Mem	When a Realm accesses an IPA within its PAR but it is `Unknown`, the Realm will copy the data from a special initialization buffer in memory to exclusive memory before accessing the IPA. This can only be done once per granule. The buffer is populated before the Realm is activated, and cannot be changed once it has been activated.
Mem	When a Realm accesses an IPA outside of its PAR, it will directly access memory, not exclusive memory.
Reg	On any trap from a Realm to the RMM, a Realm exposes the contents of various exclusive system registers, marking them `Unknown`, and marks various timer-related exclusive registers `Unknown`.
Reg	If a trap is due to system register emulation, a Realm will mark a specified exclusive GPR as `Unknown`.
Reg	If a trap is due to a hypercall, a Realm will expose and mark the seven exclusive GPRs `r0-r6` used for parameter passing as `Unknown`.
Reg	If a trap is due to an RMM call, a Realm will expose and mark the four exclusive GPRs `r0-r3` used for parameter passing as `Unknown`.

Table 3: Declassification rules.

We can then use this ideal system model with declassification to verify that RMM guarantees Realm confidentiality and integrity. The key is to establish a simulation relation in which all machine states are equivalent between the ideal and real systems and show that, at any step in the two systems satisfying the simulation relation, the same data is obtained when accessing memory or registers. This involves proving a one-to-one mapping of data between the two systems. With declassification, the mapping will change such that a different mapping will be used depending on whether the data is declassified or not. For example, if a granule within a Realm's PAR is not declassified, we will want to show that accessing that granule in non-exclusive memory in the real system correponds to accessing it in exclusive memory in the ideal system to get the same data. On the other hand, if a granule within a Realm's PAR is declassified, because its contents were initialized from an NS granule, we will want to show that first accessing that granule in non-exclusive memory in the real system correponds to accessing it in non-exclusive memory in the ideal system since the respective exclusive memory is initially `Unknown` so the data is first copied from non-exclusive to exclusive memory.

5 CCA Implementation and Verification

We used VIA to verify an early prototype implementation of CCA firmware, which includes both RMM and EL3M as

Description	LOC	Description	LOC
Machine model	1.4K	RMM refinement proofs	6.1K
Lock proof	1.7K	Top-level specification	1.1K
EL3M layer specifications	.2K	Ideal secure system model	.2K
EL3M refinement proofs	.9K	Security simulation proofs	3.4K
RMM layer specifications	4.4K	Permutation condition proofs	1.2K
Total			**20.6K**

Table 4: Lines of Coq code for verifying CCA firmware.

described in Section 3. The verification outcomes, including the discovery of several latent bugs, were confirmed by Arm's development team and used to further improve the firmware implementation. RMM contains 3.2K lines of code (LOC) in C and .3K LOC in assembly. The runtime critical parts of EL3M contain .1K LOC in C and .7K LOC in assembly; all of the C code is for updating the GPT. All RMM and EL3M code is verified, except for the portion of assembly code for initialization (.1K LOC in RMM and .5K LOC in EL3M). For remote attestation, RMM also uses functions provided by a crypto library, which was not verified, though a verified crypto library could be ported and used instead [42, 61].

Table 4 shows our proof effort, measured in LOC in Coq. 45 abstraction layers were used. The bottom layer machine model is based on VRM's Promising Arm model [57] to model Arm's relaxed memory. Another layer was used to verified the spinlock implementation on the relaxed memory model and lift it to an SC model. We verify the EL3M implementation refines its layered specification through three layers. On top of that, we verify the RMM implementation refines its layered specification through 39 layers. The top-level specification reflects RMM's interface, combining both RMM and EL3M functionality. Another layer defines the ideal secure system model. We verify that the top-level specification simulates the ideal secure system model.

5.1 Concurrent Multi-level Page Tables

The most challenging refinement proofs were for verifying RMM's RTT implementation. RTT primitives use hand-over-hand locking to synchronize access to dynamically allocated 4-level page tables, allowing fine-grain concurrent operation on different page table levels. This required nine layers. We leverage mover oracle queries and log refinement, discussed in Section 4.1, to refine all of RMM's page table operations to atomic operations, verifying the correctness of hand-over-hand locking in a real system for the first time.

Figure 8 visualizes the proof. Since acquiring a lock is a RightMover, releasing a lock is a LeftMover, and reading the page table entry is both a LeftMover and RightMover, we can reorder mover oracle queries to refine the procedure of walking the page table until acquiring the lock of `T1` into an atomic step. We group the local CPU events into a single higher-level aggregate "walk until level 1" event. Similarly, we can group events together from creating a level 1 table into a "create level 1 table' event, and destroying a level 1 table

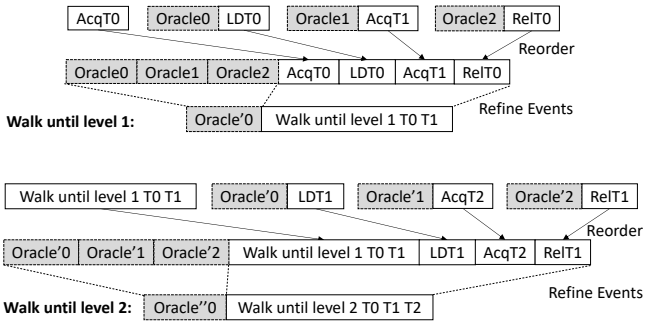


Figure 8: Proving atomicity for page table operations.

into a "destroy level 1 table" event.

We then refine the procedure of walking the page table until acquiring the lock of `T2` into an atomic step. We first prove that "walk until level 1" is a RightMover because any subsequent events at this layer from other CPUs can be reordered with it, i.e., "create level 1 table", "destroy level 1 table", "walk until level 1", and `acq`/`rel`/`LD`/`ST` events for `T2` and `T3` level tables. A "create level 1 table" from other CPUs is irrelevant to the local "walk until level 1" because it can only create other level 1 tables and cannot overwrite `T1` since RMM only allows creating a table that does not exist yet. Events "destroy level 1 table" and "walk until level 1" from other CPUs are irrelevant because they cannot hold `T1`'s lock so can only access other level 1 tables, not `T1`. Other events are also irrelevant because they do not manipulate `T0` and `T1` tables. Therefore, "walk until level 1" is a RightMover and all subsequent mover oracle queries can be reordered before it. Thus, we refine "walk until level 2" into an atomic step, as shown in the bottom of Figure 8. In a similar fashion, we prove "walk until level 2" to be a RightMover and refine the steps of "walk until level 3." Continuing in this manner, we eventually refine all RTT operations into atomic steps.

Proving RTT operations to be atomic allows us to prove desired properties about RMM's RTT management. The key property to prove is that each non-empty entry in the RTTs, including both intermediate entries pointing to lower-level RTTs and leaf mappings, uses a unique delegated granule. This prevents page remapping attacks while still allowing fine-grained access to the RTTs for improved performance. The proof is straightforward because every operation on an RTT entry is proved to be atomic, only the PA of a delegated granule is used to populate a previously empty RTT entry, and each such granule is guaranteed to be unused and zeroed. Once a granule is used for an RTT entry, its state changes from delegated to RTT or Data, preventing it from being used for other RTT entries. By using mover oracle queries and log refinement, we complete the first proof of hand-over-hand locking in a real system, and the first proof of a system with fully dynamically allocated shared page tables.

5.2 Relaxed Memory

We prove permutation conditions as discussed in Section 4.2 to verify the proofs hold on Arm relaxed memory hardware. Veri-

fying CCA firmware only requires six permutation conditions, the `RECLIST` empty condition discussed in Section 4.2, and five conditions previously introduced by VRM, namely (1) NO-BARRIER-MISUSE, (2) TRANSACTIONAL-PAGE-TABLE, (3) SEQUENTIAL-TLB-INVALIDATION, (4) WRITE-ONCE-KERNEL-MAPPING, and (5) MEMORY-ISOLATION. NO-BARRIER-MISUSE requires that barriers are correctly placed. We verified that all lock acquisitions have `acquire` memory semantics and all lock releases have `release` memory semantics. We also proved that memory accesses to shared objects outside critical sections have `release` semantics so that they cannot be reordered, preserving program ordering and SC behavior.

TRANSACTIONAL-PAGE-TABLE requires that shared page table writes within a critical section are transactional. This ensures that page table writes will not result in any behavior on relaxed memory hardware that cannot be produced on an SC model. In RMM and EL3M, each critical section contains at most one page table write, so they are obviously transactional.

SEQUENTIAL-TLB-INVALIDATION requires that a page table unmap or remap be followed by a TLB invalidation, with a barrier between them. This precludes relaxed memory behavior in TLB management code. There are no remaps in RMM or EL3M. We verified that all page table unmaps are followed by a TLB invalidation with a barrier between them.

WRITE-ONCE-KERNEL-MAPPING requires that if RMM or EL3M's own page tables are shared, they can only be written once—only empty page table entries can be modified. This precludes relaxed memory behavior due to out-of-order reads of these page tables. For EL3M, this holds as it uses a statically reserved hardcoded page table shared across all CPUs that is never changed after booting. For RMM, although its kernel page table is shared across all CPUs and can be changed, we prove that it is logically partitioned into two tables, as discussed in Section 3. We prove one table is shared but never changed once initialized, and the other table is not shared because it is statically divided into per-CPU ranges private to each CPU.

MEMORY-ISOLATION requires that the memory space accessible by RMM and EL3M is partially isolated with Realms and NS hypervisors. This ensures that any relaxed memory behavior of Realms or NS hypervisors cannot be propagated to RMM or EL3M. We verify that Realms and the hypervisor will only access `Data` and `NS` granules. Realms' memory accesses are managed by RTTs, We prove RTTs will only map `Data` granules and `NS` granules. A hypervisor's memory accesses are controlled by the GPT. We prove all delegated granules are in the Realm PAS state in the GPT so the hypervisor cannot access them. We further prove that RMM and EL3M behavior do not rely on what Realms or the hypervisor may do with `Data` or `NS` granules. We prove EL3M never accesses memory other than its own, RMM will not access the contents of `Data` granules, and whenever RMM accesses `NS` granules, it may obtain arbitrary data because the hypervisor can make arbitrary changes to the data. Thus, we show RMM's proof on SC does not rely on the concrete implementation of Realms or NS hypervisors.

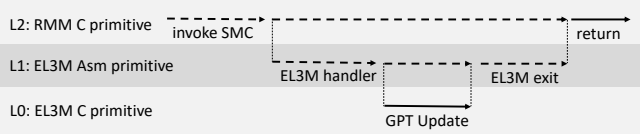


Figure 9: Verify RMM and EL3M GPT update operations. Solid arrows represent C code and dashed arrows represent assembly code.

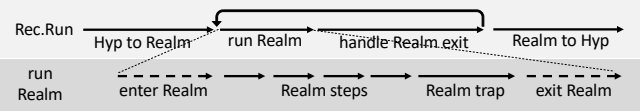


Figure 10: Verify `REC.Run` and its inner `run_realm` loop. Solid arrows represent C code and dashed arrows represent assembly code.

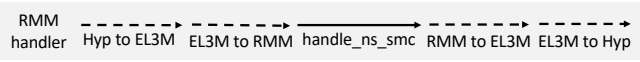


Figure 11: Verify `rmm_handler` in the top layer. Solid arrows represent C code and dashed arrows represent assembly code.

5.3 C and Assembly Code Integration

Another key aspect of the refinement proofs was verifying the interactions between RMM and EL3M, RMM and Realms, and RMM and the hypervisor, which required the C and assembly code integration techniques discussed in Section 4.3. For RMM and EL3M, we verified the correctness of GPT updates. Figure 9 shows how to verify a C primitive in RMM which issues an SMC to EL3M to update the GPT. Layer `L0` verifies the C code for EL3M's GPT operations. Layer `L1` verifies EL3M's assembly code handler, which handles traps from RMM and calls the GPT operations in C. Finally, layer `L2` verifies the C code in RMM that traps to EL3M's assembly code handler.

For RMM and Realms, we verified `REC.Run`, which runs a VCPU of a Realm and required five layers. Figure 10 shows this C primitive, which calls the `run_realm` assembly code primitive, which restores the Realm's VCPU contexts and enters the Realm. We proved that all GPRs are correctly restored such that there is no information leakage from RMM to the Realm through registers with `Unknown` values.

For RMM and the hypervisor, we verified the RMM handling of RMI calls from the hypervisor. Figure 11 shows when the hypervisor invokes an RMI call, it traps to EL3M first, then jumps to RMM and calls the C function `handle_ns_smc` to execute the RMI call. Eventually, RMM returns to EL3M and then the hypervisor. We proved that when returning to the hypervisor, there is no information leakage to the hypervisor through GPRs with `Unknown` values.

5.4 Security

We prove that the real system specified by the RMM top-level specification simulates the ideal system model with declassification, as discussed in Section 4.4. We discuss the simulation relation in three parts: all machine states except for `Data`

granules, CPU registers, and VCPU contexts stored in REC granules (**Rel 1**), Data granules (**Rel 2**), and CPU registers and VCPU contexts (**Rel 3**). Each relation is proved by induction, in which we assume the relation is initial true at machine boot and prove that it is preserved during RMM, hypervisor, and Realm execution so that the same data is obtained when accessing memory or registers in both real and ideal systems.

We prove that **Rel 1** is preserved during execution and all data accessed from memory is the same. **Rel 1** concerns NS granules, delegated granules, and granules containing Realm metadata including RTTs, none of which involve declassification. We prove two invariants: (1) all RTTs only map IPAs within the respective Realm's PAR to Data granules and IPAs outside its PAR to NS granules; and (2) the GPT only labels NS granules in the NS PAS while all delegated granules are labeled in the Realm PAS. The first invariant ensures that Realms will only access Data and NS granules, and the former will not affect **Rel 1**. The second invariant ensures that the hypervisor can only access NS granules. Since Realms and the hypervisor access NS granules in the same non-exclusive memory in both real and ideal systems, they will obtain the same data. All other granules for **Rel 1** can only be accessed by RMM. Since RMM accesses NS and other granules in the same non-exclusive memory in both real and ideal systems, it will obtain the same data; the VCPU contexts that are part of REC granules are excluded here and considered in **Rel 3**.

We prove that **Rel 2** is preserved during execution. The invariant above ensures that the hypervisor cannot access Data granules, and we prove that RMM does not access Data granules, so **Rel 2** is preserved for both the hypervisor and RMM. Data granules are only accessed by Realms. From **Rel 1**, the RTTs must be the same in both real and ideal systems. If an RTT maps an ipa within a Realm's PAR to a Data granule at host physical address hpa, the Realm will access the same data at exclusive memory ipa in the ideal system as at hpa in the real system, so **Rel 2** is preserved. To ensure that an hpa cannot be mapped to ipas in different Realms, we prove an invariant that if an RTT maps ipa to hpa, then the Data granule at hpa inversely maps to (Realm, ipa). Because there is a one-to-one mapping for each Data granule to (Realm, ipa), any changes at hpa can only be observed by the specific Realm at the specific ipa as is the case in the ideal system, so **Rel 2** is preserved for all other data. If an an ipa within a Realm's PAR is Unknown, the Realm will access the same data at non-exclusive memory hpa in the ideal and real system, so **Rel 2** is preserved.

We prove that **Rel 3** is preserved during execution. We prove if a Realm's VCPU V is running, its register r in the real system equals the corresponding exclusive register r if not Unknown or the non-exclusive register r if Unknown in the ideal system. We prove if a Realm's VCPU V is not running, V's REC context of r in the real system equals the corresponding exclusive register r if not Unknown or the V's REC context of r if Unknown in the ideal system. In the ideal system, Realm's register data is always stored in the exclusive registers except for those being declassified. Exclusive registers are not involved in context switches. We then prove that RMM indeed correctly saves and restores Realms' VCPU contexts, so that **Rel 3** is preserved.

Finally, we note that our simulation proofs between the real system and ideal secure system model verify Realm confidentiality and integrity without even trusting the correctness of the RMM or EL3M specifications. The proofs only need to trust the specification of the ideal secure system model, which encodes the declassification rules and consists of only .2K LOC in Coq. Furthermore, as shown in Table 3, the declassification rules only allow a Realm to disclose its data in two ways, by writing NS granules outside of its PAR or via the eight GPRs used for hypercalls, making the security policy formalization easy to understand.

5.5 Bugs Found

We identified several bugs in the CCA firmware prototype implementation during verification. Through refinement proofs, we detected common bugs such as incorrect boundary checking for some variables and misuse of locks; some locks were released without previously holding them. More importantly, verification of C and assembly code integration identified a serious security bug that neither EL3M nor RMM clear the caller-saved registers when returning to the hypervisor. These registers may carry RMM's private execution states and leak information. For example, RMM saves and restores Realms' VCPU contexts, and some contexts may remain in caller-saved registers and leak to the untrusted hypervisor. Another bug identified was in the REC execution handler. The hypervisor provides an NS granule to communicate entry and exit information with RMM. RMM locks and checks that the given granule is indeed an NS granule, accesses its contents, unlocks the granule, and enters the Realm. However, when exiting from the Realm, RMM did not lock and check the granule state before accessing it. This may lead to RMM unexpectedly receiving a Granule Protection Fault (GPF) from the hardware when accessing the granule using the NS PAS, if the granule was delegated by another CPU. This could lead to a denial of service of RMM or have worse consequences if GPF handling was not properly implemented in RMM.

Through permutation condition proofs, we identified an RMM bug that REC.Destroy does not implement "counter−−" with the release semantics (instruction (e) in Figure 4) such that it can be reordered with (d) on Arm's relax hardware. This may cause Realm.Destroy to wrongly set the RECLIST to be reusable before REC.Destroy clears it because when counter is zero, all RECs in the list should have been destroyed, which was not true due to this relaxed memory bug.

Through security proofs, we identified an RMM bug that allows the hypervisor to create two Data granules for the same memory address of a Realm. Thus, RMM can unmap one Data granule from an IPA of a Realm and map another Data granule to the same IPA, violating the Realm integrity guarantee,

because the Realm could observe a change in Realm data not caused by a Realm memory access.

5.6 CCA KVM

CCA provides a standard application binary interface (ABI) to allow hypervisors to communicate their intents to RMM via RMI commands, which is suitable for adoption by commodity hypervisors. However, existing hypervisors do require some modifications to use CCA to support Realm VMs. Regardless of whether a hypervisor is modified to use CCA, it cannot compromise the confidentiality and integrity of Realms. Without modifications, existing hypervisors cannot run Realm VMs, but can still run non-Realm VMs.

We modified the Linux KVM hypervisor to use CCA, which we refer to as CCA KVM. The modifications involved roughly 3K LOC in C to KVM, including .5K LOC for RMI commands, .4K LOC for handling exits from Realms, .8K LOC for creating and destroying Realms, and 1.1K LOC for stage 2 page table management using RMI commands. The modifications also required roughly .5K LOC in C to QEMU, mostly related to VM boot, initialization, and exit handling. Finally, roughly 40 LOC in C of modifications to the virtio driver in the Linux guest kernel were required so that it uses a bounce buffer to communicate I/O data with the hypervisor. This is needed because the ring buffer normally used by the virtio driver in the VM is in memory not accessible to the hypervisor when using Realms. Our experience with KVM indicates that the modifications required for a commodity hypervisor to use CCA are quite modest and involve changes to a very small percentage of its existing codebase.

6 Performance Evaluation

We have run the CCA software stack, including RMM, EL3M, and modifications to the Linux KVM hypervisor to use Realms, on an Arm Fast Model which implements the Realm Management Extensions (RME) CPU architecture. The Fast Model is a valid software emulation of the CPU architecture, allowing us to demonstrate that the CCA software stack provides the desired security guarantees and system functionality. However, Fast Models do not provide any cycle accurate measure of real performance and are too slow to run real application workloads. While CCA will be available in Armv9-A, Armv9-A hardware is not yet available.

To provide a preliminary measure of CCA performance, we have ported the CCA software prototype to run on currently available Arm hardware, an Arm N1 System Development Platform (N1SDP) [5] with an Armv8.2-A Neoverse N1 SoC. This version of EL3M is based on the the Trusted Firmware-A (TFA) codebase. The N1SDP does not provide GPT or Realm world hardware, so it cannot enforce the security guarantees of Realms, but we can use it to mimic the performance costs of Realms by modifying the EL3M code. Context switching between NS and Realm worlds is mimicked by modifying EL3M to switch between two separate contexts within NS world. EL3M is further modified to support the RMI as well as handle GPT update requests from RMM. We did not include EL3M code that controls GPT registers as they do not exist on the N1SDP, but all data written to the GPT memory can be done, although without any effect.

This setup necessarily will have some performance differences from real CCA hardware, but it provides a useful approximation of actual Realm performance. The cost of GPT checks by CCA hardware are not included since no GPT hardware is available, but are expected to exhibit good caching behavior and will not affect the relative performance of VMs versus Realm VMs since they apply equally in NS and Realm worlds. The cost of some hypervisor operations, such as those that require exiting to userspace, will be overly conservative as controlling timer interrupt behavior requires those operations to write to the Arm Generic Interrupt Controller (GIC) on the N1SDP which is slow, whereas real CCA hardware will have system registers that can be used by RMM to achieve the same functionality. Finally, the current prototype lacks support for directly injecting virtual interrupts without hypervisor intervention, which is expected to be available in future CCA hardware.

We ran both microbenchmark and application workloads in VMs on unmodified KVM and CCA KVM in Linux 5.12 on the N1SDP, which has two dual-core 2.6 GHz Neoverse N1 CPUs, 6 GB RAM, a 240 GB SATA3 SSD and a Intel 82574L 1 Gbps NIC. We used QEMU 4.2.0 [8] to run VMs, with the modifications discussed in Section 5.6 to support CCA KVM. VMs were run using KVM or CCA KVM with 4 cores and 1 GB RAM with the VM capped at 2 VCPUs and 512 MB RAM; VCPUs were pinned to individual cores. VHOST networking was used and virtual block storage devices were configured with `cache=none` [28, 38, 56]. Arm VHE [6, 17, 18] was used for all measurements. For client-server workloads, clients ran on an x86 machine with a 16-core Intel Xeon E5-2690 2.9 GHz CPU, 378 GB RAM and an Intel I350 1 Gbps NIC, connected to the N1SDP via a Linksys LGS108 1 Gbps switch.

6.1 Microbenchmarks

We ran KVM unit tests [39], which execute common micro-level hypervisor operations, plus an additional system register access microbenchmark, as listed in Table 5. For each test, we ran it 2^{16} times and report the average latency. Table 6 shows the microbenchmark measurements in nanoseconds for unmodified KVM and CCA KVM. The measurements show that the security benefits of CCA design do come with a performance cost on most micro-level hypervisor operations, because the cost of transitioning between a VM and the hypervisor is much more expensive on CCA KVM than unmodified KVM, which is most clearly shown for Hypercall.

Hypercall simply traps from the VM to the hypervisor in EL2 and returns for KVM, but involves additional operations

Name	Description
Hypercall	Trap from a VM to the hypervisor and return to the VM immediately. Measures base transition cost of hypervisor operations.
I/O Kernel	Trap from a VM to the emulated interrupt controller in the host OS kernel and return to the VM. Measures cost of accessing I/O devices supported in kernel space.
I/O User	Trap from a VM to read the device ID of virtio mmio device then return to the VM. Measures base cost of operations that access I/O devices emulated in user space.
Virtual IPI	Issue virtual IPI to another VCPU on a different CPU. Measures time from sending virtual IPI until receiving VCPU handles it.
Sysreg	Trap from a VM to emulate access to system register `ID_AA64PFR0_EL1` in the hypervisor and return to the VM. Measures system register access cost.

Table 5: Microbenchmarks.

for CCA KVM: (1) trap from VM in EL1 to RMM in EL2; (2) map NS granule to copy exit info to NS world, unmap granule; (3) trap from RMM to EL3M in EL3; (4) save Realm context, restore NS context; (5) exception return from EL3M to hypervisor in EL2; (6) trap from hypervisor to EL3M in EL3; (7) save NS context, restore Realm context; (8) exception return from EL3M to RMM in EL2; (9) map NS granule to copy entry info from NS world, unmap granule; (10) map and read data in REC and RD granules, unmap granules; (11) exception return from RMM to VM in EL1. The additional operations result in Hypercall costing an additional 1.5 μs on CCA KVM than vanilla KVM. Roundtrip transitions between RMM and the hypervisor take roughly 700 ns, and roundtrip transitions between the VM and RMM take roughly 60 ns. Saving and restoring system registers when transitioning between the VM and RMM takes roughly 200 ns per transition, or 400 ns total. The four map/unmap operations take roughly 100 ns each, 400 ns total. The remaining roughly 250 ns is due to other bookkeeping code, including saving and restoring GPRs and error checking.

I/O Kernel and I/O User include the same transition from the VM to the hypervisor and back as the Hypercall, so they also require more than 1.5 μs to execute on CCA KVM than vanilla KVM. Although the difference between CCA KVM and vanilla KVM is roughly 1.5 μs for I/O Kernel, the difference for I/O User is roughly 2.3 μs. This is because on the N1SDP, CCA KVM must write to the GIC when going to userspace, which is quite slow and takes an extra 800 ns.

Virtual IPI is more expensive on CCA KVM versus vanilla KVM because it involves multiple transitions between a VM and the hypervisor. Sending the virtual IPI involves the source vCPU writing to a system register, causing a trap to the RMM, which forwards the operation to the hypervisor (1). The hypervisor issues a physical IPI to the CPU running the destination vCPU, then returns to the source vCPU (2). The physical IPI causes an exit from the destination vCPU (3). On taking this exit, the hypervisor detects that there is a pending virtual IPI, and returns to the destination vCPU (4). Of these four transitions, approximately two occur in parallel, so the cost is roughly twice that of a Hypercall on CCA KVM for the transitions, plus the cost of the actual operation. Because Hypercall is much faster for unmodified KVM, its Virtual IPI cost is not dominated by the transition cost between VM and hypervisor.

Benchmark	Hypercall	I/O Kernel	I/O User	Virtual IPI	Sysreg
KVM	362	549	1,761	1,806	437
CCA KVM	1,865	2,060	4,049	4,324	70

Table 6: Microbenchmark performance (ns).

The one microbenchmark that is much faster on CCA KVM than KVM is Sysreg. Accessing system registers is roughly 5 times as expensive on KVM versus CCA KVM. On CCA KVM, RMM handles this register access directly without returning to the hypervisor. RMM's system register trap handling mechanism is simpler than KVM's because it does not need to support KVM's more general hypervisor functionality that requires synchronizing accesses to hypervisor-related data structures and additional conditional checks.

6.2 Application Benchmarks

We next ran the application benchmarks listed in Table 7 to measure performance on more realistic workloads. We also ran the workloads on native hardware running the same kernel to provide a baseline for comparison, restricting the system to use 2 CPUs and 512 MB RAM to provide a comparable configuration to the VMs. For each platform, we ran each workload 50 times and measured the average, worst, and best performance.

Figure 12 shows the average performance for each benchmark for unmodified KVM versus CCA KVM, with error bars indicating worst and best performance. Performance was normalized to average native execution on the N1SDP hardware; lower is better. Unlike microbenchmark performance, the application benchmark performance shows that CCA KVM and KVM have much more modest performance differences on more realistic workloads.

CCA KVM has less than 8% overhead versus unmodified KVM for most workloads, but in the worst case, overhead was 18% for MongoDB, an I/O intensive workload. The I/O intensive workloads have higher overhead for a couple reasons. The main reason is because the VM exits more frequently, so the cost of exits has a more significant impact on performance. Exits are more expensive on CCA KVM as shown by the Hypercall microbenchmark results in Table 6, in which an exit to the hypervisor costs an extra 1.5 μs. If there are many exits as will be case for I/O intensive workloads, this additional cost can become significant. For example, Memcached incurs roughly a million VM exits to the hypervisor. This results in roughly 1.5 s of additional overhead, or .75 s of overhead per core if the exits are split evenly across cores for a VM with 2 VCPUs. Memcached takes 9 s to run on vanilla KVM, so this is 8% overhead due to the extra latency for exits on CCA KVM, which roughly matches the actual overhead measured for Memcached on CCA KVM versus vanilla KVM.

A secondary reason is because CCA KVM needs to use a bounce buffer while vanilla KVM does not. CCA KVM needs a bounce buffer to support virtio because Realm memory is protected from the hypervisor. KVM uses the default virtio

Name	Description
Apache	Apache server v2.4.41 handling 100 concurrent requests via TLS/SSL from remote ApacheBench [1] v2.3 client, serving the index.html of the GCC 7.5.0 manual.
Hackbench	Hackbench [54] using Unix domain sockets and 20 process groups running in 500 loops.
Kernbench	Compilation of the Linux kernel v4.18 using allnoconfig for Arm with GCC 9.3.0.
Memcached	Memcached v1.5.22 handling requests from a remote memtier [51] v1.2.11 client with default parameters.
MongoDB	MongoDB server v3.6.8 handling requests from a remote YCSB [14] v0.17.0 client running workload A with 16 concurrent threads and operationcount=500000.
MySQL	MySQL v8.0.27 running sysbench v1.0.11 with 32 concurrent threads and TLS encryption.
Redis	Redis v4.0.9 server handling requests from a remote redis-benchmark client (redis-tools v5.0.7) [52] running `GET/SET` with 50 parallel connections and 12 pipelined requests.

Table 7: Application benchmarks.

mechanism to directly access VM memory, so it does not require bounce buffers and does not need to perform the additional data copying. Since KVM can also be configured to use a bounce buffer, we also measured KVM with this configuration to isolate the impact of using a bounce buffer on performance. The overhead with versus without a bounce buffer was negligible in most cases, but in the worst case as high as 3-4% for the more disk I/O intensive workloads, MongoDB and MySQL.

We expect the overheads for I/O intensive workloads on real CCA hardware to be less than what we measured on the N1SDP hardware. Exits are expected to occur less frequently on real CCA hardware when support for direct virtual interrupt injection is added. Exits that go to userspace are expected to cost less on real CCA hardware as the expensive GIC writes required for N1SDP hardware will be eliminated, though this was not a dominant factor in our results with the use of VHOST networking. This cost can be further mitigated by using device passthrough instead of paravirtual I/O, which will largely avoid these exits and their associated performance overhead. Support for Realm device passthrough will be added to future CCA hardware. Overall, our measurements indicate that CCA's security guarantees can be delivered with acceptable performance overheads for real application workloads.

7 Related Work

Hardware-enforced trusted execution environments have become an important feature of major computer architectures. Arm TrustZone [4] can be used to statically partition and isolate a memory region in Secure world, but most implementations only support a small number of such memory regions, limiting its scalability. Intel Software Guard Extensions (SGX) [33] can be used by application developers to protect userspace memory from other programs, including a potentially malicious OS or hypervisor. SGX is not suitable for securing VMs.

AMD Secure Encrypted Virtualization (SEV) [2] and Intel Trust Domain Extensions (TDX) [32] provide protection at the

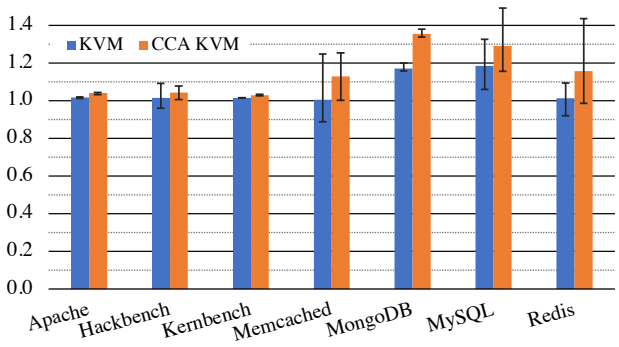


Figure 12: Application benchmark performance.

level of VMs with similar threat models to CCA. The initial version of SEV ensured confidentiality by encrypting VM memory at runtime, but did not ensure memory data integrity, which has been utilized as an attack vector such that a compromised hypervisor can tamper with or steal private VM data [31,40,47,48, 60]. Secure Nested Paging (SNP) [3] now provides the previously missing integrity protection capability. SEV-SNP allows an untrusted hypervisor to directly manage NPTs, but checks accesses against a reverse map table, an additional data structure managed by a security co-processor. In contrast, Intel TDX runs a TDX module in a privileged SEAM (Secure-Arbitration Mode) root CPU mode. The firmware manages NPTs used by protected VMs in response to requests issued by the untrusted hypervisor. Unlike CCA, the security of SGX, SEV, SEV-SNP and TDX relies on complex implementations in unverified microcode and firmware [12, 15]. They are difficult to update, either to patch security flaws or introduce new features.

Komodo [23] draws on ideas from SGX, but is implemented as a software monitor in verified Arm assembly code on top of TrustZone instead of requiring hardware to support complex enclave-manipulation instructions. This avoids hardware complexity and enables deployment of new enclave features independently of CPU upgrades. Komodo does not support multiprocessor execution, largely due to the challenge of verifying low-level concurrent code. CCA retains the advantages of Komodo's approach by relying on a verified software monitor to implement Realms, but supports verified VM protection and multiprocessor execution.

The idea of retrofitting a commodity hypervisor so that its security guarantees are enforced by a small trusted core was first explored by SeKVM [41–43, 57]. SeKVM was the first to show how this retrofitting approach, known as microverification, makes it possible to verify that a commodity hypervisor guarantees the confidentiality and integrity of VMs. CCA allows hypervisors to be modified to support Realm VMs, whose confidentiality and integrity are protected by a verified monitor, reminscient of SeKVM. While SeKVM uses existing Arm hardware, CCA introduces new hardware mechanisms that protect VMs from untrusted software running in both NS and Secure world, and allow hypervisors to make full use of Arm virtualization features such as VHE for better

performance. Furthermore, CCA firmware is designed to support a higher degree of scalability and concurrent operation by allowing data races, leveraging fine-grain synchronization, and enabling the hypervisor to provide fully dynamic memory allocation for all VM-related metadata.

While verifying CCA firmware required new VIA verification techniques, many of them build on previous work. Various concurrent systems have been verified, including CertiKOS [26, 27, 45], SeKVM, and CMAIL using CSPEC [10]. CertiKOS and SeKVM support sequential reasoning with a local CPU model and encapsulate other CPUs' behavior by rely/guarantee conditions, but do not support reordering using mover types, making proving hand-over-hand locking infeasible. Although hand-over-hand locking can theoretically be proved using rely/guarantee reasoning [58], the approach is not machine-checkable or scalable to a real system like RMM. CSPEC provides proof patterns with mover types, but lacks a local CPU model and does not verify C code; it offers little help for RMM code not reducible by movers (e.g. `REC.Destroy` in Figure 4) that still need rely/guarantee reasoning to verify. VIA builds on CertiKOS, SeKVM, and CSPEC to combine a local CPU model with mover types.

Some programs have been previously verified on relaxed memory hardware. Armada [46] supports verifying programs on the x86-TSO memory model, but their approach of verifying the entire program on a relaxed memory model has not been shown to scale to real systems such as RMM. VRM [57] instead allows proofs on an SC model to hold on relaxed memory hardware by ensuring certain conditions hold, making possible the verification of SeKVM, the first machine-checked proof for concurrent systems software on Arm relaxed memory hardware. VIA generalizes VRM to arbitrary non-DRF programs.

Verifying programs with both C and assembly code has been done to varying degrees, but none support bidirectional calls between them. seL4 [37] verifies C code, but its assembly code is unverified. CertiKOS relies on a verified x86 C compiler to verify assembly primitives invoking C primitives by compiling the invoked C primitives into assembly primitives, but cannot verify C primitives that invoke assembly primitives. Since no verified Arm C compiler exists, this approach cannot be used for CCA. SeKVM verifies C and Arm assembly code separately, but does not link the proofs, in part because no verified Arm C compiler exists. Komodo is written entirely in assembly code which is then verified, but this is difficult to scale to a large system as it is hard to write and maintain a large codebase in assembly. Ironclad [29] conducts verification at the assembly level by compiling programs in a high-level language down to assembly. This is also difficult to scale as it is harder to verify the much larger generated assembly code than the original high-level language implementation. VIA allows most proofs to be done at the C level while verifying interactions between C and assembly code are safe.

Noninterference has been frequently used to prove information-flow security [16, 23, 29, 34, 42, 49, 55], but cannot be applied to RMM given the definition of data integrity and confidentiality supported by Realms. While most of these approaches rely on some static partitioning of memory to simplify their noninterference proofs, RMM imposes no such scalability limitations. The ideal/real simulation paradigm has been used to verify information-flow security of a simple 750 LOC two-user uniprocessor separation kernel without page tables [22], but we show for the first time how it can be applied in the presence of declassification to verify data confidentiality and integrity of a real system that supports modern multiprocessor and MMU hardware with page tables.

8 Conclusions

Arm CCA is the first confidential compute architecture backed by verified firmware that is correct and secure. CCA introduces Realms, secure execution environments that protect the confidentiality and integrity of VMs against untrusted system software such as hypervisors. Realms are made possible by hardware support for Realm world, a new physical address space for Realms inaccessible to untrusted system software, and a firmware monitor that runs in Realm world to control CCA hardware to secure and manage Realms, including handling requests from untrusted hypervisors to create Realms, run Realms, and allocate memory to Realms. This design maintains compatibility with the Arm architecture without introducing complex hardware mechanisms by relying on firmware, and avoids complexity in the firmware by relying on existing hypervisors to provide virtualization functionality.

We formally verified CCA firmware, demonstrating the feasibility of relying on trustworthy firmware for the security guarantees of the architecture. We introduced various verification techniques to make it possible to verify for the first time concurrent firmware with data races running on relaxed memory hardware, fine-grain synchronization such as hand-over-hand locking, dynamically allocated shared multi-level page tables, and integrated C and assembly code. We also prove the security guarantees despite untrusted software being in full control of resource allocation decisions. The proof only needs to trust roughly two hundred lines of Coq specification, making the formal security guarantees easy to read and understand. CCA provides its security guarantees with only modest performance overhead compared to running VMs with the Linux KVM hypervisor without verified VM protection.

9 Acknowledgments

Andrew Baumann and Charles Garcia-Tobin provided helpful comments on earlier drafts. This work was supported in part by Arm, OPPO, an Amazon Research Award, a Guggenheim Fellowship, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080. Ronghui Gu is the Founder of and has an equity interest in CertiK.

References

[1] ab, The Apache Software Foundation. http://httpd.apache.org/docs/2.4/programs/ab.html, April 2015.

[2] Advanced Micro Devices. Secure Encrypted Virtualization API Version 0.16. https://support.amd.com/TechDocs/55766_SEV-KM%20API_Spec.pdf, February 2018.

[3] Advanced Micro Devices. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, January 2020.

[4] ARM Ltd. ARM Security Technology Building a Secure System using TrustZone Technology. https://documentation-service.arm.com/static/5f212796500e883ab8e74531, April 2009.

[5] ARM Ltd. Arm Neoverse N1 Core Technical Reference Manual. https://developer.arm.com/documentation/100616/0400/, April 2019.

[6] ARM Ltd. Virtualization Host Extensions. https://developer.arm.com/documentation/102142/0100/Virtualization-Host-Extensions, January 2019.

[7] ARM Ltd. Procedure Call Standard for the Arm® 64-bit Architecture (AArch64). https://github.com/ARM-software/abi-aa/releases/download/2022Q1/aapcs64.pdf, April 2022.

[8] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track (FREENIX 2005)*, pages 41–46, Anaheim, CA, April 2005.

[9] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, February 2017.

[10] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 306–322, Carlsbad, CA, October 2018.

[11] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 243–258, Huntsville, ON Canada, October 2019.

[12] Anrin Chakrabortid, Reza Curtmola, Jonathan Katz, Jason Nieh, Ahmad-Reza Sadeghi, Radu Sion, and Yinqian Zhang. Cloud Computing Security: Foundations and Research Directions. *Foundations and Trends in Privacy and Security*, 3(2):103–213, February 2022.

[13] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 431–447, Santa Barbara, CA, June 2016.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*, pages 143–154, Indianapolis, IN, June 2010.

[15] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, January 2016. https://ia.cr/2016/086.

[16] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 648–664, Santa Barbara, CA, June 2016.

[17] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, pages 304–316, Seoul, South Korea, June 2016.

[18] Christoffer Dall, Shih-Wei Li, and Jason Nieh. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, pages 221–234, Santa Clara, CA, July 2017.

[19] Christoffer Dall and Jason Nieh. KVM/ARM: Experiences Building the Linux ARM Hypervisor. Technical Report CUCS-010-13, Department of Computer Science, Columbia University, June 2013.

[20] Christoffer Dall and Jason Nieh. Supporting KVM on the ARM Architecture. *LWN Weekly Edition*, pages 18–22, July 2013.

[21] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 333–347, Salt Lake City, UT, March 2014.

[22] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS 2013)*, pages 223–234, Berlin, Germany, November 2013.

[23] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*, pages 287–305, Shanghai, China, October 2017.

[24] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, and Haozhong Zhang. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL 2015)*, pages 595–608, Mumbai, India, January 2015.

[25] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan Wu, Vilhelm Sjöberg, and David Costanzo. Building Certified Concurrent OS Kernels. *Communications of the ACM*, 62(10):89–99, September 2019.

[26] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, pages 653–669, Savannah, GA, November 2016.

[27] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 646–661, Philadelphia, PA, June 2018.

[28] Stefan Hajnoczi. An Updated Overview of the QEMU Storage Stack. In *LinuxCon Japan 2011*, Yokohama, Japan, June 2011.

[29] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 165–181, Broomfield, CO, October 2014.

[30] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. Formal Specification and Verification of Data Separation in a Separation Kernel for an Embedded System. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 346–355, Alexandria, Virginia, October 2006.

[31] Felicitas Hetzelt and Robert Buhren. Security Analysis of Encrypted Virtual Machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2017)*, pages 129–142, Xi'an, China, April 2017.

[32] Intel Corporation. Intel Trust Domain Extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, October 2014.

[33] Intel Corporation. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, May 2021.

[34] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing Browser Security Guarantees through Formal Shim Verification. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, pages 113–128, Bellevue, WA, August 2012.

[35] C. B. Jones. Tentative Steps toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, October 1983.

[36] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and Liveness of MCS Lock—Layer by Layer. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS 2017)*, pages 273–297, Suzhou, China, November 2017.

[37] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2009)*, pages 207–220, Big Sky, MT, October 2009.

[38] KVM contributors. Tuning KVM. http://www.linux-kvm.org/page/Tuning_KVM, May 2015.

[39] KVM contributors. KVM Unit Tests. http://www.linux-kvm.org/page/KVM-unit-tests, August 2020.

[40] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1257–1272, Santa Clara, CA, August 2019.

[41] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1357–1374, Santa Clara, CA, August 2019.

[42] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (IEEE S&P 2021)*, pages 1782–1799, San Francisco, CA, May 2021.

[43] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, pages 3953–3970, Vancouver, BC Canada, August 2021.

[44] Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[45] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, December 2019.

[46] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*, pages 197–210, London, UK, June 2020.

[47] Mathias Morbitzer, Manuel Huber, and Julian Horsch. Extracting Secrets from Encrypted Virtual Machines. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY 2019)*, pages 221–230, Dallas, TX, March 2019.

[48] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security (EuroSec 2018)*, pages 1–6, Porto, Portugal, April 2018.

[49] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (IEEE S&P 2013)*, pages 415–429, San Francisco, CA, May 2013.

[50] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 225–242, Huntsville, ON Canada, October 2019.

[51] Redis Labs. Memtier Benchmark. https://github.com/RedisLabs/memtier_benchmark, January 2018.

[52] Redis Labs. Redis Benchmark. https://redis.io/docs/reference/optimization/benchmarks/, March 2022.

[53] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc-12.1.0/gcc.pdf, May 2022.

[54] Rusty Russell. Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, January 2008.

[55] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 287–305, Carlsbad, CA, October 2018.

[56] SUSE. Performance Implications of Cache Modes. https://www.suse.com/documentation/sles11/book_kvm/data/sect1_3_chapter_book_kvm.html, September 2016.

[57] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP 2021)*, pages 866–881, Virtual Event, Germany, October 2021.

[58] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving Correctness of Highly-Concurrent Linearisable Objects. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006)*, pages 129–136, New York, NY, March 2006.

[59] Alexander Van't Hof and Jason Nieh. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, Carlsbad, CA, July 2022.

[60] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (IEEE S&P 2020)*, pages 1483–1496, San Francisco, CA, May 2020.

[61] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 1789–1806, Dallas, TX, October 2017.

[62] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 259–274, Huntsville, ON Canada, October 2019.

DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols

Jianan Yao
Columbia University

Runzhou Tao
Columbia University

Ronghui Gu
Columbia University

Jason Nieh
Columbia University

Abstract

Distributed systems are complex and difficult to build correctly. Formal verification can provably rule out bugs in such systems, but finding an inductive invariant that implies the safety property of the system is often the hardest part of the proof. We present DuoAI, an automated system that quickly finds inductive invariants for verifying distributed protocols by reducing SMT query costs in checking invariants with existential quantifiers. DuoAI enumerates the strongest candidate invariants that hold on validate states from protocol simulations, then applies two methods in parallel, returning the result from the method that succeeds first. One checks all candidate invariants and weakens them as needed until it finds an inductive invariant that implies the safety property. Another checks invariants without existential quantifiers to find an inductive invariant without the safety property, then adds candidate invariants with existential quantifiers to strengthen it until the safety property holds. Both methods are guaranteed to find an inductive invariant that proves desired safety properties, if one exists, but the first reduces SMT query costs when more candidate invariants with existential quantifiers are needed, while the second reduces SMT query costs when few candidate invariants with existential quantifiers suffice. We show that DuoAI verifies more than two dozen common distributed protocols automatically, including various versions of Paxos, and outperforms alternative methods both in the number of protocols it verifies and the speed at which it does so, including solving Paxos more than two orders of magnitude faster than previous methods.

1 Introduction

The world relies on distributed systems, but these systems are increasingly complex and hard to design and implement correctly. To address this problem, developers are starting to turn to formal verification techniques to prove the correctness of distributed systems [11, 20, 35]. This involves formally verifying that desired safety properties hold for the distributed protocol. A safety property is an invariant that should hold true at any point in a system's execution. It ensures the protocol does not reach invalid or dangerous states. For example, the safety property for a distributed lock protocol [11] is that no two nodes in the system hold a lock at the same time. The proof requires finding an invariant that implies the safety property, then proving that it is inductive. An invariant is inductive if it holds for all initial states of the system, and is preserved on all valid transitions so that it holds for any reachable state of the system.

Unfortunately, finding an inductive invariant is often the hardest part of the proof [21]. Invariants can be expressed as logical formulas consisting of universal ($\forall$) and existential ($\exists$) quantifiers with a certain number of variables, and a set of logical relations among the variables. Recent work has focused on automating the process of finding an inductive invariant, but has various limitations. I4 [21] was the first to automate the process, but provides no guarantee that it can find the inductive invariant and does not work for invariants with existential quantifiers. Our previous work DistAI [38] provides speed advantages over I4 and a guarantee of finding an $\exists$-free inductive invariant if one exists, but also does not work for invariants with existential quantifiers. FOL-IC3 [13] was the first to handle existential quantifiers, but is inefficient due to its heavy use of expensive SMT queries. It often fails to find invariants for protocols that can be solved by other approaches such as I4 and DistAI. SWISS [10] can successfully find an inductive invariant for Paxos, but does not work for more complex protocols such as stoppable Paxos [27]. It fails or is much slower than I4 and DistAI for many protocols without existential quantifiers.

We present DuoAI, an automated system to quickly find inductive invariants for verifying distributed protocols, with and without existential quantifiers, including complex versions of Paxos. Even though a distributed protocol may be used in very large systems, its invariants are likely to be concise, as protocols need to be designed and understood by humans to be correct. As a result, DuoAI operates in formula space and considers smaller formulas first to enumerate candidate invariants, which are then checked by an SMT solver. Formula size is defined by a maximum number of quantified variables (a variable and its quantifier $\forall$ or $\exists$) and relations. If DuoAI does

not succeed with smaller formulas, it increases the formula size and repeats the process until an inductive invariant is found. Although the formula space within a given formula size is finite, checking all possible invariants for even a modest size formula is prohibitively expensive, especially since SMT solvers are particularly inefficient at checking invariants with existential quantifiers. It is crucial to avoid too many SMT queries and SMT queries that are too complex. Based on this observation, DuoAI introduces and combines new techniques that avoid the limitations of SMT solvers in checking invariants with existential quantifiers.

First, DuoAI runs protocol simulations at various instance sizes and logs the reached protocol states, which we call *samples*. Instance size refers to the size of distributed system (number of nodes, packets, etc.) running the protocol. These simulations are fast to execute. DuoAI directly checks candidate invariants against the samples, pruning those that do not hold to reduce the number of invariants checked by an SMT solver. To do this systematically, DuoAI introduces the *minimum implication graph*, which for a given invariant, shows all its implied weaker invariants. It then selects the *strongest* candidate invariants in the graph that hold for the samples.

Second, DuoAI combines the strongest candidate invariants with the safety property and feeds them to an SMT solver to check if the conjunction is inductive. If the check fails, it *monotonically weakens* the invariants using the graph and repeats the process until an inductive invariant is found. If the number of candidate invariants is not too large and most are required in the final solution, this method will be effective at reducing the number of SMT queries by feeding all of the candidate invariants to the SMT solver at once.

Third, DuoAI feeds the strongest candidate universal invariants, those without existential quantifiers, from the graph to an SMT solver to check if the conjunction, without the safety property, is inductive. If the check fails, it monotonically weakens the invariants using the graph, only considering candidate universal invariants, and repeats the process until the conjunction is inductive. We call this set of inductive $\forall$-only invariants the *universal core*. It then strengthens the *universal core* by iteratively adding a small subset of the strongest candidate invariants with existential quantifiers from the graph until the conjunction with the safety property is inductive. If the number of candidate invariants with existential quantifiers is large and most are not in the final solution, this method will be effective at avoiding too complex SMT queries, because it only feeds a few invariants to the SMT solver each time.

DuoAI runs these two methods for refining candidate invariants in parallel, a top-down refinement that *weakens* the candidates and a bottom-up refinement that *strengthens* the candidates, returning the result from the method that succeeds first. We prove that both methods are guaranteed to find the inductive invariant that proves the desired safety property, but they may have very different running times and resource requirements depending on the distributed protocol being verified. Using both methods together provides the best of both worlds in addressing the inefficiencies of SMT solvers.

```
1  type value
2  type quorum
3  type node
4
5  relation vote(N1:node, N2:node)
6  relation voted(N:node)
7  relation leader(N:node)
8  relation decided(N:node, V:value)
9  relation member(N:node, Q:quorum)
10 axiom forall
       Q1, Q2. exists N. member(N, Q1) & member(N, Q2)
11
12 after init {
13     voted(N) := false;
14     vote(N1, N2) := false;
15     leader(N) := false;
16     decided(N, V) := false;
17 }
18
19 action cast_vote(n1: node, n2: node) = {
20     require ~voted(n1);
21     vote(n1, n2) := true;
22     voted(n1) := true;
23 }
24
25 action become_leader(n: node, q: quorum ) = {
26     require forall N. member(N, q) -> vote(N, n);
27     leader(n) := true;
28 }
29
30 action decide(n:node, v: value) = {
31     require leader(n);
32     require forall V. ~decided(n, V);
33     decided(n, v) := true;
34 }
35
36 invariant decided(N1,V1) & decided(N2,V2) -> V1=V2
```

Figure 1: The simplified consensus protocol written in Ivy. Capitalized variables are implicitly quantified. For example, Line 16 means $\forall N:node, V:value.\ decided(N,V) := false$. "~" stands for negation.

We evaluated DuoAI using 27 widely-used distributed protocols in a head-to-head comparison against other approaches, including I4, DistAI, FOL-IC3, and SWISS. DuoAI outperforms all of the other approaches in terms of both the number of protocols for which it finds an inductive invariant and the speed at which it does so. DuoAI solves Paxos more than two orders of magnitude faster than any other approach, and is the only system that can solve more complex versions of Paxos including multi-Paxos, stoppable Paxos, and fast Paxos.

2 Overview

We use a simplified consensus protocol as an example to show how DuoAI works. Figure 1 shows the protocol written in Ivy [28], a language and tool for specifying, modeling, and verifying distributed protocols built on top of the Z3 SMT solver. Each node can vote for another node to be the leader, and when a node receives votes from a quorum of nodes, it can become the leader and decide on a value. The protocol state at any moment is represented by five relations (Lines 5-9). $vote(n_1,n_2)$

indicates whether node n_1 has voted for node n_2. $voted(n)$ indicates whether node n has ever casted a vote. $leader(n)$ indicates if n is the leader among nodes. $decided(n,v)$ indicates whether node n has decided on value v. $member(n,q)$ indicates if node n belongs to quorum q, where each quorum is a set of nodes. The axiom (Line 10) dictates a property of the member relation: any two quorums of nodes must have at least one node in common. After initialization (Lines 13-16), the protocol can non-deterministically transition from one state to another as described by the three actions *cast_vote*, *become_leader*, and *decide* (Lines 19-34). For example, $cast\_vote(n_1,n_2)$ lets a node n_1 vote for another node n_2, under the precondition that n_1 has not voted before (Line 20). Then the protocol will transition to a new state where $vote(n_1,n_2) = true$ and $voted(n_1) = true$. Finally, the safety property (Line 36) encodes the desired property of correctness of the protocol that the system cannot decide on two different values.

The safety property is an invariant of the protocol, but is not inductive as taking an action from a state satisfying the safety property may result in a new state that breaks the safety property. To verify the protocol, we need four additional invariants:

$$\forall N_1,N_2 : node.\ vote(N_1,N_2) \rightarrow voted(N_1) \quad (1)$$

$$\forall N_1,N_2,N_3 : node.\ vote(N_1,N_2) \wedge vote(N_1,N_3) \rightarrow N_2 = N_3 \quad (2)$$

$$\exists Q : quorum.\ \forall N_1,N_2 : node.$$
$$leader(N_1) \wedge member(N_2,Q) \rightarrow vote(N_2,N_1) \quad (3)$$

$$\forall N : node.\ V : value.\ decided(N,V) \rightarrow leader(N). \quad (4)$$

The first invariant says that if a node has voted for another node, then it must be recorded as *voted* in the protocol. The second says that one node cannot vote for two different nodes. The third says that a leader must be endorsed by a quorum of nodes. More specifically, we can find a quorum Q that every node N_2 in the quorum must have voted for the leader N_1. The fourth says that only a leader can decide on a value. The conjunction of the four invariants and the safety property is inductive.

To find this inductive invariant, DuoAI simulates the protocol using different instance sizes and logs the samples. It then builds a minimum implication graph, a small fragment of which is shown in Figure 2. The full graph for simplified consensus has over 35K nodes and 170K edges. Nodes represent formulas and edges represent implication between formulas. A stronger formula will have a directed edge to an implied weaker formula. DuoAI enumerates possible candidate invariants following the graph and adds it to the candidate invariant set if it holds on the samples. For example, DuoAI checks the root node in Figure 2 and it does not hold on the samples. DuoAI then checks its implied weaker formulas, the two nodes in the second layer, iteratively going down the graph. For the simplified consensus protocol, enumeration ends with 19 candidate invariants, including equivalent forms of Eq. (1), (2), (3), and (4).

After enumeration, DuoAI runs top-down and bottom-up refinement in parallel. Top-down refinement feeds all candidate invariants and the safety property to Ivy to see if their conjunction is inductive. For simplified consensus, the conjunction is inductive, so no further weakening is required. Bottom-up refinement feeds all ∀-only invariants from the initial candidate set to Ivy then weakens them until the set of invariants is itself inductive, but may not imply the safety property. For simplified consensus, this universal core includes three invariants Eq. (1), (2), and (4). DuoAI then tries to search a small number of ∃-included invariants to add to the universal core along with the safety property so that the resulting set is inductive. DuoAI uses counterexamples from Ivy to guide the search for additional invariants and eventually identifies invariant (3) for the simplified consensus protocol, forming an inductive invariant set. For simplified consensus, top-down refinement succeeds more quickly than bottom-up refinement.

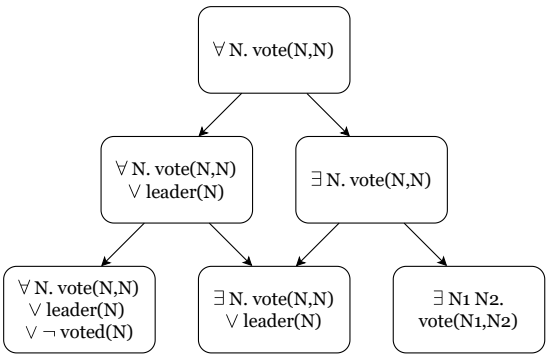


Figure 2: Fragment of the minimum implication graph for the simplified consensus protocol.

3 Minimum Implication Graph

The backbone of DuoAI is the minimum implication graph, which encodes implication relations among formulas. The graph is used to determine the order of formulas to be enumerated, and how invariants are weakened. We present formulas in prenex normal form, where the quantified variables, called the prefix, appear at the beginning of the formula followed by quantifier-free relations, called the matrix. The matrix is required to be in disjunctive normal form (DNF). For simplicity, here we only consider predicate symbols with equality. The methods can be extended to uninterpreted functions in the same manner as DistAI [38].

A formula P is strictly stronger than Q if $P \Rightarrow Q$ and $Q \not\Rightarrow P$. For two formulas $P,Q \in \mathcal{S}$, where $\mathcal{S}$ is a finite formula search space, there is a directed edge from P to Q in the minimum implication graph if and only if P is strictly stronger than Q and there is no formula R which is strictly weaker than P while strictly stronger than Q. For example, the fragment of the minimum implication graph in Figure 2 includes three formulas:

$$\forall N.\ vote(N,N) \quad (5)$$

$$\exists N.\ vote(N,N) \quad (6)$$

$$\exists N_1,N_2.\ vote(N_1,N_2) \quad (7)$$

Eq. (5) $\Rightarrow$ (6) since if $vote(N,N)$ is true for all N, there must exist some N for which it is true. Eq. (6) $\Rightarrow$ (7) since if $vote(N,N)$ is true for some N, there must exist some $N_1 = N_2$ for which $vote(N_1,N_2)$ is true. Because Eq. (5) $\Rightarrow$ (6) $\Rightarrow$ (7), there is an edge from Eq. (5) to (6), an edge from Eq. (6) to (7), but no edge from Eq. (5) to (7), because Eq. (6) is between them.

DuoAI defines the search space $\mathcal{S}$ as all formulas in disjunctive normal form for a given set of quantified variables and formula size. The formula size is defined by four parameters: *max_exists* sets the maximum number of existentially quantified variables, *max_literal* sets the upper bound of the total number of literals in the formula, *max_and* sets the maximum number of literals connected by AND, and *max_or* sets the maximum number of conjunctions connected by OR.

The minimum implication graph has two important properties as stated in Lemmas 1 and 2:

Lemma 1. *The minimum implication graph is a directed acyclic graph (DAG).*

Proof. Suppose there is a cycle $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_k \rightarrow P_1$. The edges $P_1 \rightarrow P_2, \ldots, P_{k-1} \rightarrow P_k$ imply that $P_1 \Rightarrow P_2, \ldots, P_{k-1} \Rightarrow P_k$. From the transitivity of $\Rightarrow$ we know $P_1 \Rightarrow P_k$. Since there is an edge from P_k to P_1, we know $P_1 \not\Rightarrow P_k$, a contradiction. □

Lemma 2. *For any $P,Q \in \mathcal{S}$, there is a path from P to Q in the minimum implication graph if and only if $P \Rightarrow Q \wedge Q \not\Rightarrow P$.*

Proof. We first prove the "if" direction by induction on the number of formulas in S that are strictly weaker than P while strictly stronger than Q. For the base case, if there are zero such formulas, then by definition there is an edge from P to Q. Next we prove the induction step. Suppose for any $P,Q \in S$, if $P \Rightarrow Q \wedge Q \not\Rightarrow P$, and there is no more than n formulas that are strictly weaker than P while strictly stronger than Q, then there is a path from P to Q. Now consider the case that there are $n+1$ formulas that are strictly weaker than P while strictly stronger than Q. Let R be one of the $n+1$ formulas. We know $P \Rightarrow R \wedge R \not\Rightarrow P$, and there can be no more than n formulas that are strictly weaker than P while strictly stronger than R. By the induction hypothesis, there is a path from P to R. In the same manner, we can show there is a path from R to Q. Then we concatenate the two paths and get a path from P to Q.

Next we prove the "only if" direction. If there is a path from P to Q, Let $P, F_1, F_2, ..., F_k, Q$ be the path. We know $P \Rightarrow F_1, \ldots, F_k \Rightarrow Q$, so $P \Rightarrow Q$. We prove $Q \not\Rightarrow P$ by contradiction. Suppose $Q \Rightarrow P$, then $P \Leftrightarrow Q$, so there must be an edge from F_k to P. This forms a cycle $P,F_1,...,F_k,P$, a contradiction to Lemma 1. □

To build the minimum implication graph, we need to determine the "root" nodes in the graph, that is, formulas with no predecessors since they cannot be implied by any other formula, and how to find their successors. In DuoAI, a formula $P \in S$ is added to the set of root nodes if it falls into one of two cases:

1. P has no $\exists$-quantified variable and no logical OR. For example:
$$\forall N : node.\ vote(N,N) \wedge leader(N). \tag{8}$$
2. P has unique $\exists$-quantified variables and no logical OR. For example:
$$\exists N_1,N_2 : node.\ N_1 \neq N_2 \wedge vote(N_1,N_2). \tag{9}$$

Intuitively, if a formula has an $\exists$, then by changing it to a $\forall$, we can get a stronger formula. If a formula has a logical OR, then by removing the OR and any literals followed by it, we can get a stronger formula. So in general, a root formula should have no $\exists$ and no OR, such as Eq. (8). There is one exception, represented by Eq. (9). At first sight Eq. (9) has a predecessor $\forall N_1,N_2 : node.\ N_1 \neq N_2 \wedge vote(N_1,N_2)$. However, this formula is a contradiction because $\forall N_1,N_2 : node.\ N_1 \neq N_2$ cannot be true. The minimum implication graph does not include tautologies and contradictions, so Eq. (9) itself is a root formula.

Starting from the root nodes, DuoAI incrementally builds the minimum implication graph. For formulas $P,Q \in S$, DuoAI adds an edge from P to Q if the shapes of P and Q fall into one of five cases:

1. P and Q share the same matrix. Q replaces the $\forall$-quantified variables of one type with $\exists$-quantified variables. For example:
$$P = \forall N : node, V : value.\ \neg decided(N,V)$$
$$Q = \exists N : node.\ \forall V : value.\ \neg decided(N,V).$$
2. P and Q share the same prefix. Q has one less ANDed literal than P. For example:
$$P = \text{Eq.}(8) \qquad Q = \text{Eq.}(5).$$
3. P and Q share the same prefix. Q has one more ORed conjunction than P. For example:
$$P = \forall N : node.\ vote(N,N)$$
$$Q = \forall N : node.\ vote(N,N) \vee (voted(N) \wedge leader(N)).$$
DuoAI requires that the ORed conjunction be maximal, which means it contains the maximum number of literals for the search space. The conjunction $voted(N) \wedge leader(N)$ in Q is maximal if $max\_and = 2$ or $max\_literal = 3$. For example, $Q' = \forall N : node.\ vote(N,N) \vee voted(N)$ also adds one more ORed conjunction from P, but DuoAI does not add an edge from P to Q', because Q is strictly stronger than Q'.
4. Starting from P, Q projects two $\forall$-quantified variables of the same type into one variable. For example:
$$P = \forall N_1,N_2 : node.\ vote(N_1,N_2) \vee leader(N_1)$$
$$Q = \forall N : node.\ vote(N,N) \vee leader(N).$$
5. Starting from Q, P projects two $\exists$-quantified variables of the same type into one variable. For example:
$$P = \text{Eq.}(6) \qquad Q = \text{Eq.}(7).$$

The graph constructed in this way may differ slightly from the exact minimum implication graph due to equivalent formulas. For example, formulas $\forall X.\ p(X) \vee (\neg p(X) \wedge q(X))$ and $\forall X.\ p(X) \vee q(X)$ fall into the second case, so there is an edge in the constructed graph. However, the two formulas are equivalent so there is no edge in the exact graph. We call the graph constructed by DuoAI an *approximate minimum implication graph*, whose properties are formalized in Lemmas 3, 4, and 5:

Lemma 3. *The approximate minimum implication graph is a directed acyclic graph (DAG).*

Proof. For all of the five cases, we can show that for formulas along any path in the approximate minimum implication graph, there exists one function that is strictly increasing, so there can be no cycle. For example, this is true for the function $(\#\exists\text{-}variables) - (\#\forall\ variables) + (max\_and * (\#\vee)) - (\#\wedge)$, where # denotes "the number of" (e.g., $(\#\vee)$ is the number of logical OR in a formula). □

Lemma 4. *For any $P,Q \in \mathcal{S}$, there is a path from P to Q in the approximate minimum implication graph only if $P \Rightarrow Q$.*

Proof. From the transitivity of $\Rightarrow$, we only need to show that if there is an edge from P to Q in the approximate minimum implication graph, then $P \Rightarrow Q$. This can be proved by showing $P \Rightarrow Q$ holds in each of the five cases. The first three cases are trivial. For the fourth case, in general $P = \ldots \forall X_1 X_2 \ldots matrix(X_1, X_2)$ and $Q = \ldots \forall X_1 \ldots matrix(X_1, X_1)$. Let $P' = \ldots \forall X_1 X_2 \ldots X_1 = X_2 \rightarrow matrix(X_1, X_2)$, then $P \Rightarrow P' \Leftrightarrow Q$. Similarly, for the fifth case, $P = \ldots \exists X_1 \ldots matrix(X_1, X_1)$ and $Q = \ldots \exists X_1 X_2 \ldots matrix(X_1, X_2)$. Let $Q' = \ldots \exists X_1 X_2 \ldots X_1 = X_2 \wedge matrix(X_1, X_2)$, then $P \Leftrightarrow Q' \Rightarrow Q$. □

Lemma 5. *For any formula $P \in S$ that is not a tautology or a contradiction, there exists a directed path from a root node $Q \in S$ to P in the approximate minimum implication graph.*

Proof. We prove this by construction. For a $\exists$-free formula P, if it includes no logical OR, then it is a root formula itself. Otherwise, we find the root formula Q by removing all but one ORed conjunctions. Starting from Q, we can iteratively apply the second and third cases to add conjunctions and remove literals until we reach P. For a $\exists$-included formula P, if it includes unique $\exists$-quantified variables then it is a root formula itself. Otherwise, we iteratively find a predecessor by replacing $\exists$ with $\forall$ for quantified variables of each type, until the formula becomes the $\exists$-free P'. The first case guarantees that there is a path from P' to P, and we have already shown for the $\exists$-free P', there exists a path from a root node Q. Putting it together, we have a path from Q to P in the approximate minimum implication graph. □

In other words, the approximate minimum implication graph is as useful and complete as the exact graph. DuoAI uses the approximate minimum implication graph, which, for simplicity, we will continue to refer to as the minimum implication graph unless otherwise specified.

DuoAI requires that formulas in $\mathcal{S}$ must be in a decidable fragment of first-order logic. In general, satisfiability in first-order logic is undecidable [23], so an SMT solver can get stuck in infinite instantiations and never give the *sat*/*unsat* answer. DuoAI ensures that the formulas are decidable by enforcing a fixed order of types if there is quantifier alternation (i.e., alternating $\forall$ and $\exists$) [1]. If type A is ordered before type B, then for any formula, if there exists a quantified variable V of type A, any quantified variable of type B can only occur after V if there is quantifier alternation. For example, if type *node* is ordered before type *packet*, then $\forall N : node.\ \exists P : packet$ and $\exists N : node.\ \forall P : packet$ are allowed while $\forall P : packet.\ \exists N : node$ and $\exists P : packet.\ \forall N : node$ are not. DuoAI tries to infer the order of types from the protocol specification and obtains input from the user when necessary. For example, for the simplified consensus protocol, DuoAI can infer from Line 10 that type `quorum` must be ordered before type `node`, then ask the user to place type `value` in the order. Absent user input, DuoAI will try different possible orders in parallel.

4 Candidate Invariant Enumeration

Similar to DistAI [38], DuoAI first repeatedly simulates the distributed protocol using various instance sizes, and records the reached states as samples. For example, DuoAI simulates the simplified consensus protocol on concrete instances of different numbers of values, quorums, and nodes. The simulations of different instance sizes are done in parallel and yield samples of different lengths. DuoAI follows the minimum implication graph to enumerate candidate invariants, but rather than feeding all of them to an inefficient SMT solver, it checks them directly on the samples first. A correct invariant must hold on every reachable protocol state and thus on every sample. A key difference between DuoAI and DistAI is that DuoAI keeps the original variable-length samples and uses them in invariant enumeration, while DistAI projects all samples to fixed-length vectors that it calls subsamples. The problem is that DistAI is not exhaustive in its subsampling, so that a formula with existential quantifiers that holds for DistAI's subsamples may not actually hold for the original samples. DuoAI avoids this problem by effectively considering all possible subsamples that can be derived from the original samples.

Algorithm 1 shows the enumeration algorithm, in which *pending* is a queue whose elements are formulas that will be checked on the samples, *candidates* is the set of formulas that hold on all the samples and *invalidated* is the set of formulas that do not hold on at least one of the samples. Both *candidates* and *invalidated* are initially empty (Lines 2-3), and *pending* initially consists of the root nodes of the minimum implication graph, that is, formulas that cannot be implied by any other formula. In each iteration, a formula f is popped from the *pending* queue (Line 5). If one of f's ancestors in the graph has already been added to *candidates*, DuoAI will not check f on the samples or add f to the *candidates* invariants (Lines 6-7). Oth-

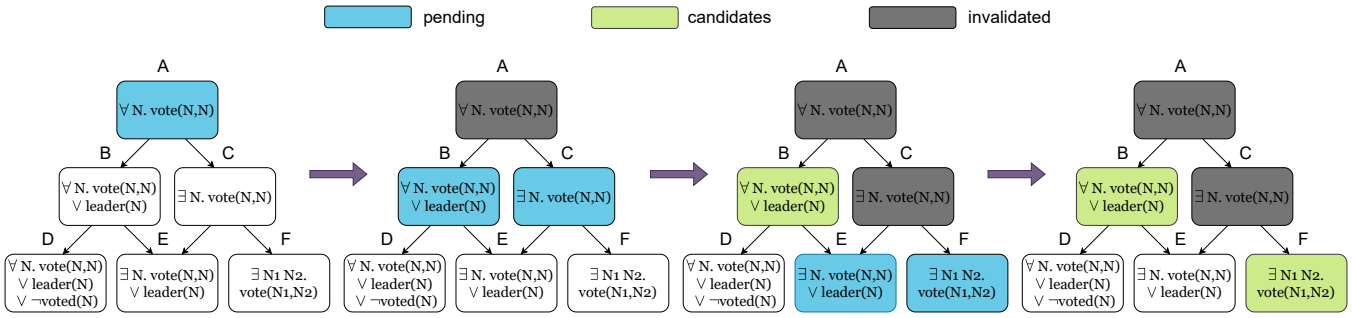


Figure 3: Invariant enumeration procedure based on the minimum implication graph. Suppose formula A and formula C do not hold on all the samples, while the other four formulas hold. Step 1: Only the root node A is in the *pending* queue. Step 2: The root node A is invalidated by the samples. We add its two successors B and C to the *pending* queue. Step 3: Formula B holds on the samples thus being added to *candidates*, while formula C is invalidated and its two successors E and F are added to the *pending* queue. Step 4: Formula E has an ancestor B which is already in *candidates*, so E is simply skipped instead of being checked on the samples. Formula F holds on the samples and is added to *candidates*.

Algorithm 1 Invariant Enumeration Algorithm

Input: Distributed protocol $\mathcal{P}$, invariant search space $\mathcal{S}$, a set of samples from protocol simulation *samples*
Output: Candidate invariants

```
1:  graph := build_minimum_implication_graph(P,S)
2:  candidates, invalidated := ∅,∅
3:  pending := graph.rootNodes
4:  while pending.notEmpty() do
5:      f := pending.dequeue()
6:      if graph.ancestors(f) ∩ candidates ≠ ∅ then
7:          continue
8:      if check_inv_holds(f, samples) then
9:          candidates := candidates ∪ {f}
10:     else
11:         invalidated := invalidated ∪ {f}
12:         for next_f ∈ graph.successors(f) do
13:             if next_f ∉ candidates and next_f ∉ invalidated
                and next_f ∉ pending then
14:                 pending.enqueue(next_f)
15: return candidates
```

erwise, DuoAI will check f on the samples and if it holds, add it to *candidates* (Lines 8-9). If f does not hold on at least one sample, DuoAI will add it to *invalidated* (Line 11), and add its successors, which are formulas weaker than f, to the *pending* queue if they have not already been added (Lines 12-14).

Figure 3 shows an example of invariant enumeration using the graph in Figure 2. DuoAI starts from the root nodes, iteratively goes down the minimum implication graph, and checks formulas against the samples. Because of this design, formulas D and E are never checked against the samples and are not added to the candidates, because their predecessor B, a formula stronger than both D and E, is already a candidate invariant. This design not only saves time checking formulas on samples, but also avoids burdening the SMT solver later with checking the inductiveness of redundant invariants. More importantly, this procedure guarantees that the resulting invariants, formulas B and F in this example, are the strongest candidate invariants that hold on the samples, which is formally stated in the following theorem:

Theorem 1. *For any correct invariant $I \in \mathcal{S}$ held by the protocol $\mathcal{P}$, at the end of invariant enumeration, either 1) $I \in candidates$, or 2) one of I's ancestors $I_{anc} \in candidates$.*

Proof. Consider three cases: 1) I has been checked on the samples, 2) I has been added to the *pending* queue but was not checked on samples, and 3) I has been never added to the *pending* queue. In the first case, since I is a correct invariant held by the protocol, it must hold on all the samples and will be added to *candidates* (Lines 8-9), so $I \in candidates$. In the second case, after I is popped from the *pending* queue, there must be an ancestor I_{anc} of I already in *candidates* (Line 6), otherwise I will be checked on the samples, so $I_{anc} \in candidates$. In the third case, we show that an ancestor $I_{anc} \in candidates$ exists. From Lemma 5, there must be a path from a root node I_0 to I, namely $I_0,I_1,...,I$. On Line 3 the root node I_0 is added to the *pending* queue. Since I_0 is added to the *pending* queue and I is not, let I_k be the last formula on the path $I_0,I_1,...,I$ that is ever added to the *pending* queue. After I_k is dequeued, there are three possible branches to take: Lines 6-7, Lines 8-9, or Lines 10-14. If it takes Lines 6-7, then there is an ancestor I_{anc} of I_k such that $I_{anc} \in candidates$. If it takes Lines 8-9, then I_k will be added to *candidates* so I_k can be the ancestor I_{anc} of I such that $I_{anc} \in candidates$. If it takes Lines 10-14, its successors will be added to the *pending* queue unless the branch condition at Line 13 evaluates to false. From our hypothesis, I_k is last formula on path $I_0,...,I_k,I_{k+1},...,I$ that is ever added to the *pending* queue. Thus, the branch condition for I_{k+1} must evaluate to false, so either $I_{k+1} \in candidates$ or $I_{k+1} \in invalidated$. However, I_{k+1} must be added to the *pending* queue before it can be added to either *candidates* or *invalidated*, a contradiction. □

Theorem 1 says that any correct invariant has either itself or

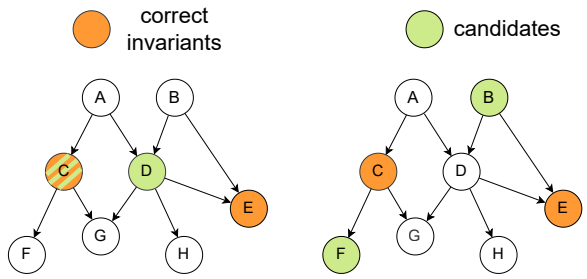


Figure 4: Possible (left) and impossible (right) candidate invariants after enumeration. Formulas C and E are correct invariants held by the protocol. It is possible that after enumeration, $candidates = \{C, D\}$ (left). Correct invariant C is in the candidate invariants itself. For correct invariant E, its ancestor D is in the candidate invariants. Theorem 1 guarantees that $candidates = \{B, F\}$ is not a possibility after enumeration, because for correct invariant C, neither itself nor its lone ancestor A is in the candidate invariants.

its ancestor (a stronger formula) in the candidate invariants. Figure 4 gives an illustration. A direct corollary is that, the set of candidate invariants after the enumeration is at least as strong as the correct invariant in $\mathcal{S}$.

5 Top-down Invariant Refinement

Based on Theorem 1, the candidate invariants can only be *too strong*, so DuoAI can monotonically weaken the candidate invariants until a correct inductive invariant is reached, which we refer to as top-down invariant refinement. Algorithm 2 shows the top-down refinement algorithm. In each iteration, DuoAI feeds the current candidate invariants to Ivy. Ivy invokes the Z3 SMT solver to check the inductiveness of each candidate invariant and the safety property. Ivy will return which invariants fail the check; if there are none, the correct inductive invariant has been found (Lines 4-5). If the safety property fails, there is no point to weaken it, and the system returns NotProvable (Lines 6-7). If one of the candidate invariants fails, DuoAI moves it from *candidates* to *invalidated* (Lines 9-10), then adds its successors (i.e., formulas that can be implied by the failed invariant) to *candidates* so long as the successor does not have a reachable ancestor in *candidates* and has not already been invalidated (Lines 12-13). An ancestor of a node is reachable if there is a path from the ancestor to the node along which no node is invalidated.

Figure 5 shows an example of top-down refinement. Suppose the current candidate invariants include formulas B and F, and by invoking the Z3 SMT solver, Ivy indicates that B is not inductive. Formulas D and E are not in *candidates*, because they can be implied by formula B which is already in *candidates*. After B is invalidated, both D and E will be added to *candidates* to let Ivy decide their inductiveness in future iterations. Alternatively, if formula F is invalidated by Ivy, no formula will be added to *candidates* because F has no successor in the minimum implication graph of search space $\mathcal{S}$.

Algorithm 2 Top-down Invariant Refinement Algorithm

Input: Distributed protocol $\mathcal{P}$, minimum implication graph *graph*, candidate invariants from enumeration *CI*
Output: Either an inductive invariant *II*, or NotProvable

```
1:  candidates, invalidated := CI, ∅
2:  while candidates.notEmpty() do
3:      failed_inv := Ivy_check(P, candidates)
4:      if failed_inv is None then
5:          return candidates
6:      else if failed_inv = safety_property then
7:          return NotProvable
8:      else
9:          candidates := candidates \ {failed_inv}
10:         invalidated := invalidated ∪ {failed_inv}
11:         for next_inv ∈ graph.successors(failed_inv) do
12:             if graph.reachable_ancestors(next_inv) ∩
                candidates = ∅ and next_inv ∉ invalidated then
13:                 candidates := candidates ∪ {next_inv}
14: return NotProvable
```

By weakening failed invariants based on the minimum implication graph rather than discarding them, DuoAI can guarantee that it never overweakens invariants to bypass the correct invariants in between. In other words, top-down refinement has a theoretical guarantee to eventually find an inductive invariant if one exists in the search space, as stated in the following theorem:

Theorem 2. *For any protocol $\mathcal{P}$ and finite search space $\mathcal{S}$, if there exists an inductive invariant $II^* \subset \mathcal{S}$ that can prove the safety property, then Algorithm 1 followed by Algorithm 2 will output such an inductive invariant II in finite time.*

Proof. The key is to prove that the while loop (Lines 2-13) maintains the following loop invariant: For any invariant $I \in II^*$, either 1) $I \in candidates$, or 2) there exists a reachable ancestor I_{anc} of I such that $I_{anc} \in candidates$. The loop invariant says that after any rounds of invariant weakening, the candidate invariants must be still at least as strong as the correct invariants. If Algorithm 2 terminates, it is impossible to have the safety property fail (Line 7). The only possibility is that a correct inductive invariant is returned (Line 5).

Theorem 1 guarantees that the loop invariant holds before entering the loop. We only need to prove that if this loop invariant holds at the beginning of round k of invariant weakening, it must still hold at the beginning of round $k+1$. This proof is done by construction for each $I \in II^*$. From the induction hypothesis, at the beginning of round k, either 1) $I \in candidates$, or 2) a reachable ancestor $I_{anc} \in candidates$. In the first case, I cannot have been invalidated during round k because $I \in II^*$, so $I \in candidates$ still holds at the beginning of iteration $k+1$. In the second case, the invalidated invariant must either be on or

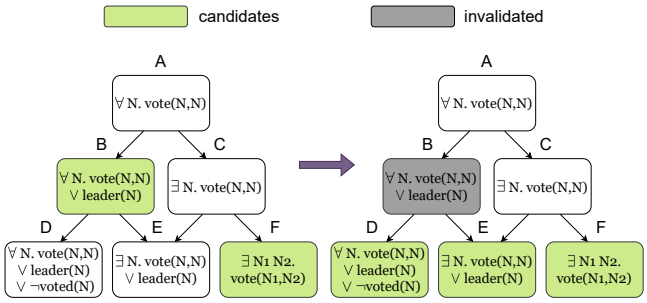


Figure 5: One round of top-down refinement. Suppose candidate invariant B fails the Ivy check. DuoAI removes B from *candidates* and adds its successors D and E to *candidates*.

not on the path from I_{anc} to I. If it is not on the path, I_{anc} remains a reachable ancestor of I and $I_{anc} \in candidates$ still holds at the beginning of iteration $k+1$. If it is on the path, let I_d be the successor of the invalidated invariant on the path. From Lines 11-13, either I_d is added to *candidates*, in which case I_d can be the new I_{anc} for iteration $k+1$, or I_d has a reachable ancestor $I_e \in candidates$, in which case we choose I_e as the new I_{anc} for iteration $k+1$. In all cases, we can find either I or a reachable ancestor I_{anc} in the candidate set, therefore the loop invariant holds.

Now we only need to prove that Algorithm 2 terminates, which follows from three observations: 1) In each loop iteration a formula is removed from *candidates* (Line 9); 2) each formula can only be added to *candidates* once (Lines 10 & 12); and 3) the formula search space $\mathcal{S}$ is finite. □

6 Bottom-up Invariant Refinement

Although top-down refinement provides a strong theoretical guarantee of finding an inductive invariant, it may take too long or run out of memory given limited computing resources if there are too many unnecessary invariants to consider. For the simplified consensus protocol in Figure 1, besides the four invariants (1)(2)(3)(4), many other invariants hold for the protocol but are unnecessary to prove the inductiveness of the safety property, for example,

$$\forall V : value, Q : quorum.\ \exists N : node.\ member(N,Q) \wedge (leader(N) \vee \neg decided(N,V)). \quad (10)$$

Invariants such as Eq. (10) do not affect the soundness of DuoAI, but they will significantly slow down the validation of candidate invariants by the SMT solver. If there are m candidate invariants, validating each invariant takes $O(m)$ time in the worst case, since the inductiveness of one invariant can depend on any other invariant, so checking all candidate invariants can take $O(m^2)$ time. Adding unnecessary invariants can increase validation time quadratically.

The key issue though is not just how many unnecessary invariants there are, but whether they have quantifier alternation (i.e., alternating ∀ and ∃), which we observe causes

Algorithm 3 Bottom-up Invariant Refinement Algorithm

Input: Distributed protocol $\mathcal{P}$, minimum implication graph *graph*, candidate invariants from enumeration *CI*
Output: Either an inductive invariant *II*, or NotProvable

Procedure 1
1: $CI_\forall := \{I | I \in CI \wedge I \text{ is } \exists\text{-free}\}$
2: $core$:= Algorithm2($\mathcal{P}$, $graph_\forall$, $CI_\forall$)
3: $noncore := CI \setminus core$
Procedure 2
4: $CE := \emptyset$
5: **for** *sub* **in** $powerset(noncore)$ **do**
6: **if** $\exists s \in CE$. invs_hold_on_state(sub,s) **then**
7: **continue**
8: result := Algorithm2($\mathcal{P}$, *graph*, $core \cup sub$)
9: **if** result = NotProvable **then**
10: $s \xrightarrow{a} s'$:= get_counterexample()
11: $CE := CE \cup \{s\}$
12: **else**
13: **return** result
14: **return** NotProvable

SMT solvers to struggle. For the Paxos protocol, a correct inductive invariant set of size 14 can be validated in less than a second. If we add 10 correct but unnecessary invariants with quantifier alternation, the validation will take 5 minutes. If we add 20 such invariants, the validation will take over 3 hours.In contrast, the chord ring maintenance protocol [21] with 149 ∀-only invariants only takes 8 seconds to validate.

However, a correct distributed protocol typically has a clear and human-understandable intuition, which leads to concise invariants [10]. This motivates our bottom-up invariant refinement algorithm shown in Algorithm 3. In essence, the algorithm tries to identify a small set of correct *and* helpful invariants that can eventually prove the safety property. §8 shows that the combination of bottom-up with top-down refinement provides fast performance for finding inductive invariants across a wide-range of protocols.

Algorithm 3 consists of two procedures. In Procedure 1, DuoAI first extracts all the ∀-only invariants from the candidate invariants (Line 1), which are guaranteed to be the strongest ∀-only invariants that hold on the samples. Then, DuoAI runs the top-down refinement algorithm (Line 2) using only the universal invariants and the universal portion of the minimum implication graph by removing all nodes representing existentially quantified formulas. The safety property is neglected in this top-down refinement. In this way, the ∀-only invariants are monotonically weakened until they become inductive, regardless of whether the safety property can be proved (it probably cannot). Recall that we call the now inductive ∀-only invariants the universal inductive core. DuoAI then puts every enumerated candidate invariant that is not in the universal inductive core

into *noncore* (Line 3). *noncore* mainly consists of formulas with existential quantifiers, but also includes ∀-only formulas that are not in the core, whose inductiveness may depend on ∃-included invariants. For example, for the simplified consensus protocol, the universal inductive core includes five candidate invariants, which are exactly the equivalent forms of Eq. (1), (2), and (4). There are 14 non-core candidate invariants, 13 of which have quantifier alternation, including Eq. (3) and (10).

Based on our observation that SMT solvers struggle with quantifier alternation, we expect *noncore* formulas will have a much higher cost of checking. Procedure 2 aims to identify a *small* subset of *noncore* to strengthen the candidate invariants, such that the conjunction of the universal inductive core and the subset (denoted as $core \cup sub$), or their weaker forms, can prove the safety property. Procedure 2 enumerates each subset *sub* of *noncore* (Line 5), and runs the monotonic weakening algorithm (Algorithm 2) on $core \cup sub$ (Line 8). If Algorithm 2 returns NotProvable (Line 9), DuoAI moves on to consider the next subset. Otherwise, Algorithm 2 outputs a correct inductive invariant (Line 13). The enumeration of subsets is conducted in increasing order of size, starting from the $\emptyset$, followed by all single formulas from *noncore*, then pairs, triples, and so on.

Whenever Algorithm 2 finds the safety property failed and reports NotProvable, Ivy returns a counterexample of inductiveness $s \xrightarrow{a} s'$ (Line 10), which means starting from a protocol state s satisfying the safety property and the candidate invariants, and taking an action a, the system reaches a new state s' where the safety property is violated.[1] If we view the samples from protocol simulation as positive samples on which the invariants must hold, then we can view these counterexample states s as negative samples which the invariants must exclude. DuoAI needs to identify and include another invariant I that does *not* hold on s, so that the counterexample $s \xrightarrow{a} s'$ can be excluded. When enumerating a subset of *noncore*, Procedure 2 first checks if the subset can exclude all counterexamples seen so far (Line 6). If there exists one counterexample state s on which all invariants in the subset hold, or in other words, the counterexample cannot be excluded, the monotonic weakening algorithm is bound to fail, because if a stronger invariant cannot exclude the counterexample, then its weaker forms cannot either. So Procedure 2 simply moves on to enumerate the next subset (Line 7).

For the simplified consensus protocol, when $sub = \emptyset$, the safety property fails and Ivy gives the counterexample $s = \{vote(n_1,n_1) = vote(n_1,n_2) = vote(n_2,n_1) = vote(n_2,n_2) = false,\ voted(n_1) = voted(n_2) = false,\ leader(n_1) = leader(n_2) = true,\ member(n_1, q) = member(n_2, q) = true,\ decided(n_2, v_1) = true,\ decided(n_1, v_1) = decided(n_1, v_2) = decided(n_2, v_2) = false\}$. Eq. (3) does not hold on s, so it can exclude this counterexample. In contrast, Eq. (10) holds on s, so the counterexample will persist even if Eq. (10) is added to the candidate set. Therefore, DuoAI will skip Eq. (10) and try Eq. (3), and run Algorithm 2 on its conjunction with the universal core, which gives a correct inductive invariant set consisting of Eq. (1)(2)(3)(4).

Although counterexamples can be used for top-down refinement, DuoAI currently does not because Ivy cannot return counterexamples in batch. When Ivy is configured to return a counterexample, it terminates once it identifies the first broken invariant. This is inefficient for top-down refinement, but for bottom-up refinement, counterexamples are only needed for the safety property, so DuoAI puts the safety property on top of other invariants and Ivy will give the desired counterexample.

Like top-down refinement, bottom-up refinement has a theoretical guarantee to eventually find an inductive invariant if one exists in the search space, as stated in the following theorem:

Theorem 3. *For any protocol $\mathcal{P}$ and finite search space $\mathcal{S}$, if there exists an inductive invariant $II^* \subset \mathcal{S}$ that can prove the safety property, then Algorithm 1 followed by Algorithm 3 will output such an inductive invariant II in finite time.*

Proof. We first prove that Algorithm 3 terminates in finite time. This directly follows from three facts: 1) $powerset(noncore)$ is a finite set so the for loop (Line 5) has a finite number of iterations; 2) In each loop iteration, there is at most one invocation of Algorithm 2 (Line 8); and 3) From Theorem 2, Algorithm 2 terminates in finite time.

To prove the soundness of Algorithm 3, we first observe that if Algorithm 3 outputs an invariant, it must be a correct inductive invariant, because the output must come from Algorithm 2, in which the output can only occur when the safety property is proved.

Now we prove that there will be an output invariant eventually. Observe that $noncore \in powerset(noncore)$ (Line 5). When $sub = noncore$, we have $CI \subset core \cup sub$, then Line 8 degenerates to Algorithm 2 in §5. From Theorem 2, we know a correct inductive invariant will be outputted. □

For both the top-down and bottom-up refinement, if NotProvable is returned, we know the protocol cannot be verified using invariants in the search space $\mathcal{S}$. DuoAI will try a larger search space by increasing either *max_literal*, *max_or*, *max_and*, or *max_exists*, or the per-domain number of quantified variables. By default, DuoAI alternates among the five in a round-robin manner. DuoAI sets the initial $max\_literal = 4$, $max\_or = max\_and = 3$, and $max\_exists = 1$ unless the safety property already involves $k \geq 2$ existentially quantified variables, in which case DuoAI sets $max\_exists = k$. DuoAI sets the initial number of quantified variables for domain T as the maximum number of variables of type T in any relation. For example, the relation `vote(N1:node,N2:node)` guarantees type `node` has at least two variables.

Because SMT solvers are much less efficient at checking invariants with existential quantifiers, and many distributed protocols are provable by ∀-only invariants [21], DuoAI runs

[1] In general, other than showing an invariant is not inductive, a counterexample may also show an invariant does not hold at the protocol initial state. But this cannot happen to the safety property, unless the protocol is wrong.

a ∀-only instance (i.e., $max\_exists = 0$) in parallel. The ∀-only instance only runs top-down refinement, as bottom-up refinement degenerates to the same top-down refinement (Line 2).

7 Optimizations Based on Mutual Implication

In using the minimum implication graph, DuoAI introduces several optimizations based on mutual implication relations among formulas. These relations further prune the search space and avoid redundant candidate invariants. DuoAI considers two kinds of mutual implication relations, 1) $P_1 \wedge P_2 \wedge ... \wedge P_k \Rightarrow Q$, and 2) $P_1 \wedge P_2 \wedge ... \wedge P_k \Leftrightarrow Q$. Although the latter is a special case of the former, DuoAI treats them differently. We refer to Q as a *conjunction implied formula* in the former and an *equivalently decomposable formula* in the latter. Since checking inductiveness has a quadratic complexity with the number of invariants, these optimizations have a significant improvement on efficiency.

Conjunction implied formulas. DuoAI identifies conjunction implied formulas to avoid redundant candidate invariants. Given a mutual implication relation $P_1 \wedge P_2 \wedge ... \wedge P_k \Rightarrow Q$, if all $P_1, P_2, ... P_k$ are already in the candidate invariants, DuoAI will mark Q as a conjunction implied formula and not add Q to the candidate invariants. Later during refinement, if one of $P_1, P_2, ... P_k$ is invalidated by Ivy, then the conjunction implied invariant Q is no longer redundant and will be added to the candidate invariants.

For example, suppose we have a disk replication protocol with the following three invariants:

$$\forall E : epoch, R : replica.\ crashed(E,R) \rightarrow \neg readable(E,R) \quad (11)$$

$$\forall E : epoch.\ \exists R : replica.\ readable(E,R) \quad (12)$$

$$\forall E : epoch.\ \exists R : replica.\ \neg crashed(E,R). \quad (13)$$

One can check that among Eq. (11)(12)(13), no formula can imply another. But the conjunction of Eq. (11) and (12) can imply Eq. (13). This is because Eq. (12) says that for every epoch E, there must be a readable replica R. Then from Eq. (11), the readable replica R cannot be crashed. Therefore, for every epoch E, there must be a replica R that does not crash, which is expressed by Eq. (13). If Eq. (11) and (12) are already candidate invariants, DuoAI will mark Eq. (13) as a conjunction implied formula and not add it to the candidate invariants.

There are many classes of mutual implication relations in first-order logic. DuoAI identifies three classes of conjunction implied formulas to prune candidate invariants; in each class, the first two formulas mutually imply the third:

1. Replace a literal with a weaker literal, as discussed in the example Eq. (11)(12)(13):

$$P_1 = \forall X.\ r(X) \rightarrow s(X)$$
$$P_2 = prefix.\ (r(X) \wedge ...) \vee ...$$
$$Q = prefix.\ (s(X) \wedge ...) \vee ...$$

2. Conjunct a literal with a weaker literal:

$$P_1 = \forall X.\ r(X) \rightarrow s(X)$$
$$P_2 = prefix.\ (r(X) \wedge ...) \vee ...$$
$$Q = prefix.\ (r(X) \wedge s(X) \wedge ...) \vee ...$$

3. "Merge" a ∀ formula and an ∃ formula:

$$P_1 = \exists X.\ r(X)$$
$$P_2 = \forall X.\ s(X) \vee ...$$
$$Q = \exists X.\ (r(X) \wedge s(X)) \vee ...$$

In all three classes, r and s can be generalized to conjunctions (e.g., $r_1(X) \wedge \neg r_2(X)$, $\neg s_1(X) \wedge s_2(X) \wedge s_3(X)$). A key advantage of this optimization is that given a finite search space, DuoAI can identify conjunction implied formulas based on invariants within that search space, even though the conjunction of invariants is not in that search space.

Equivalently decomposable formulas. DuoAI also identifies equivalently decomposable formulas to avoid redundant candidate invariants. Given a mutual implication relation $P_1 \wedge P_2 \wedge ... \wedge P_k \Leftrightarrow Q$, DuoAI will mark Q as an equivalently decomposable formula and never add Q to the candidate invariants. Later during refinement, if one of $P_1, P_2, ... P_k$ is invalidated by Ivy, Q will also be invalidated and therefore there is never any reason to consider Q further as a candidate invariant.

For example, suppose the disk replication protocol has invariant:

$$\forall E : epoch.\ \exists R_1, R_2 : replica.\ readable(E,R_1) \wedge writable(E,R_2). \quad (14)$$

There is no need to ever include such an invariant in the candidate set, because it is equivalently decomposable to invariants (15) and (16).

$$\forall E : epoch.\ \exists R : replica.\ readable(E,R) \quad (15)$$

$$\forall E : epoch.\ \exists R : replica.\ writable(E,R). \quad (16)$$

However, suppose we slightly modify invariant (14) to one of the following two formulas:

$$\forall E : epoch.\ \exists R_1 : replica.\ readable(E,R_1) \wedge writable(E,R_1) \quad (17)$$

$$\forall E : epoch.\ \exists R_1, R_2 : replica.\ R_1 \neq R_2 \wedge readable(E,R_1) \wedge writable(E,R_2). \quad (18)$$

These invariants are not equivalently decomposable to invariants (15) and (16). Take Eq. (17) as an example. One can verify that (17) $\Rightarrow$ (15) $\wedge$ (16), but (15) $\wedge$ (16) $\not\Rightarrow$ (17), because Eq. (17) requires the same replica to be both readable and writable, while for Eq. (15) and (16), it is possible to have one readable replica and a different writable replica.

DuoAI identifies if a formula is equivalently decomposable based on the structure of the formula itself by considering two classes of equivalently decomposable formulas. One class is embodied by the following lemma:

Lemma 6. *For a formula F in prenex and disjunctive normal form, we build a graph G_C for each conjunction C in F. G_C has one node for each literal, and an edge between two literals if and only if they share an existentially quantified variable. If for some C in F, the graph G_C has $k \geq 2$ connected components, then F is equivalently decomposable into k formulas.*

Proof. For simplicity, here we show the proof for $k = 2$ (i.e., the literals in C form two connected components). If there are $k > 2$ connected components, then the same analysis below will show that formula F is equivalently decomposable into k formulas.

Literals that share $\exists$-variables must be in the same connected component. So we can divide the $\exists$-variables into two sets $\{Y_1,...,Y_m\}$ and $\{Z_1,...,Z_n\}$. The first connected component can only include $\exists$-variables $Y_1,...Y_m$, while the second can only include $Z_1,...,Z_n$. Assume the quantifier prefix has shape $\forall X_1 ... X_s \exists Y_1 ... Y_m Z_1 ... Z_n$. We use $\vec{X},\vec{Y},\vec{Z}$ to denote $X_1 ... X_s$, $Y_1 ... Y_m$, and $Z_1 ... Z_n$. The proof can be generalized to any alternating $\forall/\exists$ using skolemization.

Let $f(\vec{X}, \vec{Y})$ be the disjunction of literals in the first connected component, and $g(\vec{X}, \vec{Z})$ be the disjunction of literals in the second connected component. Let $h(\vec{X},\vec{Y},\vec{Z})$ be the disjunction of all conjunctions other than C in formula F. Then formula F can be rewritten as:

$$\forall\vec{X}\exists\vec{Y}\vec{Z}.\ (f(\vec{X},\vec{Y})\wedge g(\vec{X},\vec{Z}))\vee h(\vec{X},\vec{Y},\vec{Z}). \tag{19}$$

We now show that, Eq. (19) is equivalently decomposable into:

$$\forall\vec{X}\exists\vec{Y}\vec{Z}.\ f(\vec{X},\vec{Y})\vee h(\vec{X},\vec{Y},\vec{Z}) \tag{20}$$

$$\forall\vec{X}\exists\vec{Y}\vec{Z}.\ g(\vec{X},\vec{Z})\vee h(\vec{X},\vec{Y},\vec{Z}). \tag{21}$$

It is trivial that Eq. (19) implies both Eq. (20) and (21). We now show the interesting direction — the conjunction of Eq. (20) and (21) implies Eq. (19). Suppose both Eq. (20) and (21) hold. For any $\vec{X}$, consider two cases: 1) $\exists\vec{Y}\,\vec{Z}.\ h(\vec{X},\vec{Y},\vec{Z})$. In this case Eq. (19) directly holds. 2) $\forall\vec{Y}\,\vec{Z}.\ \neg h(\vec{X},\vec{Y},\vec{Z})$. Then according to Eq. (20), $\exists\vec{Y}_1\vec{Z}_1.\ f(\vec{X},\vec{Y}_1)$. Similarly, from Eq. (21), $\exists\vec{Y}_2\vec{Z}_2.\ g(\vec{X},\vec{Z}_2)$. If we select $\vec{Y}_1$ and $\vec{Z}_2$, then we have $f(\vec{X},\vec{Y}_1)\wedge g(\vec{X},\vec{Z}_2)$, so Eq. (19) still holds. Putting the two cases together, when both Eq. (20) and (21) are true, Eq. (19) must be true. □

The proof also gives an algorithm to find the k decomposed formulas. Figure 6 shows how to apply Lemma 6 on the aforementioned formulas. For Eq. (14), the two literals $readable(E,R_1)$ and $writable(E,R_2)$ share no $\exists$-quantified variable (E is $\forall$-quantified), so there is no edge between the two literals. The graph has two connected components, so Eq. (14) is equivalently decomposable into two formulas (Eq. (15)(16)). For Eq. (17), the two literals share an $\exists$-quantified variable R_1, so there is an edge between the two literals and the graph is connected, which means Eq. (17) cannot be decomposed in this way. The same analysis can be applied to Eq. (18).

We note a corollary of Lemma 6. For an $\exists$-free formula, it is equivalently decomposable if it has any conjunction.

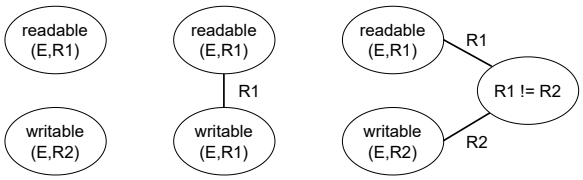


Figure 6: Checking equivalently decomposability of formulas (14)(17)(18) (from left to right).

For example, $\forall X.\ p(X) \wedge q(X)$ is equivalent with the pair $\forall X.\ p(X)$ and $\forall X.\ q(X)$. This indicates that we do not need to consider any conjunction when enumerating $\forall$-only formulas, a significant reduction in search space.

The other class of equivalently decomposable formulas that DuoAI identifies is embodied in the following:

$$\forall X_1 X_2.\ matrix(X_1,X_2) \tag{22}$$

$$\forall X_1.\ matrix(X_1,X_1) \tag{23}$$

$$\forall X_1 X_2.\ X_1 \neq X_2 \rightarrow matrix(X_1,X_2) \tag{24}$$

One can check that Eq. (22) is equivalently decomposable to Eq. (23) and (24), and will therefore not be added as a candidate invariant. In general, DuoAI only considers formulas whose leading $\forall$-quantified variables are unique. Similar optimizations have been used in DistAI [38].

8 Evaluation

Experimental setup. To demonstrate the performance of DuoAI, we implemented and evaluated DuoAI on 27 distributed protocols from multiple sources [12, 13, 21, 27, 28], including those that can only be proved by inductive invariants with $\exists$-quantifiers. The DuoAI implementation consists of 6.1K lines of C++ code for invariant enumeration and refinement, compiled by gcc 7.5.0, and 2.3K lines of Python code running with Python 3.8.10 for protocol simulation. For comparison, we also ran 6 other invariant inference tools: SWISS [10], IC3PO [8, 9], FOL-IC3 [13], DistAI [38], UPDR [12], and I4 [21]. All experiments were performed on a Dell Precision 5829 workstation with a 4.3GHz 28-core Intel Xeon W-2175, 62GB RAM, and a 512GB Intel SSD Pro 600p, running Ubuntu 18.04.

We configured the alternative invariant inference tools following their best practices. SWISS requires the user to bound the search space by specifying 4 parameters, including the number of existentially quantified variables and the number of literals in a formula. For every protocol, we use the same parameter settings as in SWISS's own evaluation [10]. IC3PO and I4 require the user to specify a finite instance size for their model checkers to work on. For IC3PO, we only specified the minimum size and the tool itself could determine how to increase the instance size. For I4, we started from the minimum size where the protocol can function and iteratively increased

the instance size upon failure (e.g., $node = 2, node = 3, ...$). FOL-IC3 provides a ∀-only mode and a default mode. We ran both and report the runtime of whichever succeeded first.

Results summary. Table 1 shows the running time in seconds for each tool on each protocol. For each protocol, we also report the number of relations and lines of code in its Ivy specification; for example, Figure 1 is a simplified version of consensus epr. The top portion of the table shows protocols provable with a ∀-only inductive invariant, while the bottom portion shows protocols that can only be proved with a ∃-included inductive invariant. We allowed each tool to spend up to an entire week trying to solve each protocol. For protocols that a tool fails to solve, we report "fail" if the tool terminated without an inductive invariant, "error" if the tool itself returned an error, "Z3 error" if the underlying SMT solver used returned an error, "memout" if the tool ran out of memory and terminated, and "timeout" if the tool did not complete within a week.

DuoAI dominates all other tools in the number of protocols it solves, solving all but 1 of the 27 protocols. SWISS cannot solve 8 protocols, FOL-IC3 and IC3PO cannot solve 9 protocols, DistAI and UPDR cannot solve 13 protocols, while I4 cannot solve 15 protocols. DuoAI is the only tool that solves all ∀-only protocols, is the only tool that solves Paxos as well as all other non-Paxos protocols with ∃ quantifiers, and is the only tool that solves 3 of the more complex Paxos variants, including multi-Paxos, stoppable Paxos, and fast Paxos. There were no protocols solvable by another tool that were not solved by DuoAI.

DuoAI also dominates all other tools in how fast it solves the protocols, solving 15 of the protocols faster than any other tool. DuoAI is faster than SWISS on all but 3 of the protocols solved by SWISS, is faster than IC3PO on all but 3 of the protocols solved by IC3PO, is faster than FOL-IC3 on all but 2 of the protocols solved by FOL-IC3, and is faster than UPDR on all of the protocols solved by UPDR. DuoAI is up to 3 orders of magnitude faster than each of these protocols. DuoAI is faster than DistAI on all but 5 of the protocols solved by DistAI, and is faster than I4 on all but 2 of the protocols solved by I4. DuoAI is up to an order of magnitude faster than either DistAI or I4. The speed differences versus DistAI and I4 appear less in part because neither could solve any of the protocols with existential quantifiers. In most cases in which DuoAI is slower than other protocols, it is by at most a few seconds.

Detailed comparison and discussion. For the protocols provable with ∀-only invariants, DuoAI is the only tool that solves all 15 protocols. On ∀-only protocols, DuoAI's ∀-only instance is similar to DistAI, without subsampling and with mutual implication optimization and parallelism in simulation. DuoAI beats DistAI on 10 protocols. Unlike DuoAI, DistAI times out on ticket lock, which we discovered is due to a bug in the implementation of its protocol simulation. Chord ring maintenance is the only protocol on which DuoAI is much slower than DistAI. DistAI only allows invariants as disjunction of literals, and implements an invariant as a `vector<int>`. In contrast, DuoAI considers invariants in disjunctive normal form, so an invariant is a disjunction of conjunction of literals, implemented as a `set<vector<int>>`. This makes invariant operation slower in DuoAI. Chord ring maintenance is the only ∀-provable protocol that takes significant time on candidate invariant enumeration so the overhead is exacerbated.

For the protocols that require invariants with ∃-quantifiers to prove, DuoAI solves 11 out of 12 protocols, more than any other tool. DuoAI only fails on vertical paxos, which other tools also fail on. DistAI, I4, and UPDR fail on all of these protocols because they can only generate ∀-only invariants. IC3PO solves 4 protocols and fails on 3 protocols, but runs out of memory on 6 protocols, because it requires model checking to infer invariants on a finite instance. For more complex protocols like fast Paxos, the model checker requires too large of an instance size. In contrast, DuoAI searches in formula space and its performance does not depend (exponentially) on instance size.

For the complex Paxos-family protocols, only SWISS also verified Paxos and flexible Paxos, though it required several hours to do so. All other tools failed on all Paxos-family protocols. In contrast, DuoAI verified Paxos and flexible Paxos in less than 2 minutes. Only DuoAI verified multi-Paxos, stoppable Paxos, and fast Paxos.

As the only other tool that solves Paxos, it is instructive to compare SWISS with DuoAI. Similar to DuoAI, SWISS also enumerates candidate invariants given a bounded search space and checks their inductiveness using the SMT solver. However, it has two fundamental differences compared with DuoAI. First, SWISS relies exclusively on the SMT solver to tell the correctness of invariants, while DuoAI also uses the samples from protocol simulation to filter out invalid invariants. As we demonstrated in §6, SMT calls can be expensive with quantifier alternation and will negatively affect performance. Second, SWISS struggles to find mutually inductive invariants, i.e., a bundle of invariants that are inductive together but none are inductive individually. This is because SWISS can only build the invariant set by adding one and only one invariant each time and keep the set inductive. In the lock server async and the sharded key-value store protocols, where mutually inductive invariants are required to prove the safety property, SWISS has to manually increase the maximum number of literals from 6 to 9. This allows the mutually inductive invariants to be conjuncted into one big invariant, but results in a much larger search space and long runtimes of 44 and 128 minutes, respectively. DuoAI enumerates candidate invariants following the minimum implication graph and generates the strongest candidate invariants. The mutually inductive invariants (or their stronger forms) are guaranteed to be in the candidate invariants together. DuoAI solved both protocols within 2 seconds.

For the vertical paxos protocol, the human-expert inductive invariants include an invariant with 8 literals. Even after the optimization based on mutual implication, it still has 7

Distributed protocol	Relations	LoC	DuoAI	SWISS	IC3PO	FOL-IC3	DistAI	UPDR	I4
chord ring maintenance	8	123	200.9	timeout	17.1	timeout[c]	58.0	Z3 error	673
consensus forall	7	55	11.9	40.3	457	1500	15.6	59.0	122
consensus wo decide	6	46	3.9	26.1	160	24.8	8.5	24.8	27.5
database chain replication	13	96	9.5	108951	4.5	559	90.3	57.6	66.6
decentralized lock	2	21	9.9	5.8	24.3	69.0	10.2	51.0	20.7
distributed lock	4	43	6.1	timeout	12856	1660	15.3	63568	195
ring leader election	3	45	3.5	14.3	memout	10.8	2.9	103	5.3
learning switch ternary	4	45	14.2	308	23.8	timeout	24.7	1334	12.9
learning switch quad	2	21	52.4	1322	63.6	timeout	372	273	memout
lock server async	5	45	1.9	2625	5.6	4.8	1.1	4.4	8.7
lock server sync	2	21	1.3	1.0	3.2	1.0	0.9	3.3	0.6
sharded key-value store	3	31	1.9	7662	5.4	9.7	1.2	3.5	error
ticket lock	5	49	23.9	fail	56.2	58.1	timeout	143	fail
toy consensus forall	4	27	1.9	5.9	3.0	5.4	3.1	3.4	9.4
two-phase commit	7	70	1.5	9.1	4.7	4.8	2.0	9.4	10.2
client server ae	4	28	1.5	5.2	2.3	355	timeout	fail	fail
client server db ae	7	48	3.1	33.7	memout[i]	4822	timeout	fail	fail
consensus epr	7	52	4.8	28.8	1118	471	timeout	fail	memout
hybrid reliable broadcast	12	120	1211.2	fail	memout[i]	931	error	fail	error
sharded kv no lost keys	3	32	2.1	1.8	4.8	3.7	timeout	fail	error
toy consensus epr	4	25	2.6	4.3	2.4	32.9	timeout	fail	fail
Paxos	9	75	60.4	16665	fail[i]	timeout	timeout	timeout	memout
flexible Paxos	10	77	78.7	28337	memout[i]	timeout	timeout	fail	memout
multi-Paxos	10	91	1549	timeout	fail	timeout	timeout	timeout	memout
stoppable Paxos	11	118	4051	error	fail	timeout	error	timeout	error
fast Paxos	12	102	26979	timeout	memout	memout	timeout	fail	error
vertical Paxos	12	120	memout	timeout	memout	memout	error	fail	error

c The SWISS authors reported that FOL-IC3 solved chord ring maintenance [10], but we found that the `chord.pyv` file they used has 3 bugs.
i The IC3PO authors [8, 9] reported that IC3PO succeeded on client server db ae (17 s), hybrid reliable broadcast (587 s), Paxos (568 s), and flexible Paxos (561 s). However, they retrofitted the protocols and manually provided clauses with quantifier alternation that could appear in the invariants, which is difficult to do without first knowing the ground-truth invariants. The 4 protocols have much simpler inductive invariants when expressed on top of these clauses, with all except the simplest, client server db ae, becoming $\exists$-free. Ivy fails when checking the invariants generated by IC3PO for Paxos and flexible Paxos. The IC3PO authors [9] imply that the invariants had to be manually checked against the human-expert invariants.

Table 1: Comparison of different tools for finding inductive invariants for 27 distributed protocols (running time in seconds).

literals. Under the minimum per-domain number of quantified variables that can encode the human-expert invariants, there are 60 predicates that can appear in the invariants. Considering their negations, the size of the invariant search space is at the magnitude of $120^7 \approx 4e14$, well exceeding the computational power of a normal workstation. In comparison, for fast paxos, the largest invariant includes 5 literals, and there are 38 predicates. The size of the search space is at the magnitude of $3e9$. For vertical Paxos, DuoAI ends with a universal core and a set of checked non-core invariants when exhausting memory. These invariants are inductive and can be utilized as hints, although they cannot imply the safety property.

As explained in §6, DuoAI runs top-down refinement, bottom-up refinement, and a $\forall$-only instance in parallel. Not surprisingly, the $\forall$-only instance generates the inductive invariants first for all 15 protocols that do not require existential quantifiers. Among the 11 protocols solved by DuoAI that require existential quantifiers, the top-down refinement gives the inductive invariants first for the 5 simpler protocols — client server ae, client server db ae, toy consensus epr, consensus epr, and sharded kv no lost keys. The bottom-up refinement also succeeded but took longer. For example, for client server db ae, there are 8 candidate invariants in *noncore*. A subset of size 3 was sufficient to prove the safety property. However, the bottom-up refinement would first enumerate and fail on all single invariants and pairs before enumerating the correct triple. This takes more than 3 times longer than top-down refinement, in which after a single round of weakening, DuoAI found an inductive invariant.

For the 6 more complicated protocols with existential quantifiers, including hybrid reliable broadcast and the 5 Paxos-family protocols solved, only the bottom-up refinement generated the inductive invariants. The top-down refinement got stuck at checking the inductiveness of the invariants. For example, for multi-Paxos, after enumeration, DuoAI has a candidate invariant set of size 615, and 581 of them have quantifier alternation. Checking inductiveness of this many formulas is a hopeless task for the Z3 SMT solver. However, to prove the safety property, only 2 of the 581 candidate invariants are needed. In the bottom-up refinement, each time only a

small subset of *noncore* invariants conjuncted with the ∀-only core are fed to Ivy, so the Z3 SMT solver could handle the candidate invariants. For Paxos, flexible Paxos, multi-Paxos, and stoppable Paxos, a subset of size 2 were sufficient, while a subset of size 6 was needed for fast Paxos. This also validates our assumption that real-world distributed protocols should have concise invariants, and should not require too many invariants with quantifier alternation to verify.

Limitations By requiring quantifier alternation to conform to a fixed order of types, DuoAI ensures that the verification condition is in a decidable fragment of first-order logic. However, without decidability, an SMT solver may still succeed. For client server db ae and hybrid reliable broadcast, the invariants written by human experts are not in a decidable fragment, yet they can be efficiently verified by the SMT solver. For both protocols, DuoAI found alternative inductive invariants within the decidable fragment. If a protocol cannot be verified in decidable logic, DuoAI will fail to prove it.

9 Related Work

Many studies [11, 17, 20, 31, 34–36] verify the correctness of distributed protocols with manually given inductive invariants. Early systems [12, 21, 29, 38] for automatically inferring inductive invariants do not work for invariants with ∃-quantifiers which are required for protocols such as Paxos, though pdH [29] can find inductive invariants for retrofitted Paxos-family protocols without ∃-quantifiers.

Recent systems consider invariants with ∃-quantifiers. FOL-IC3 [13] generates an invariant candidate that can separate a positive and negative example set, and iteratively adds more examples until the invariant is correct. It has no theoretical guarantee of success. Its heavy use of SMT queries to validate as well as synthesize invariants makes it slow in practice, timing out on even protocols with ∀-only inductive invariants.

SWISS [10] iteratively strengthens an invariant by adding small inductive formulas until the invariant is strong enough to prove the safety property. It was the first tool to automatically verify Paxos. Its inefficiency in exploring the search space and inability to infer mutually inductive invariants make it fail on several protocols solved by alternative tools.

IC3PO [8, 9] applies model checking on a finite instance similar to I4, while adding support to generalize existentially quantified invariants. The model checker functions well on small instances, but frequently exhausts memory on complex protocols that require larger instances, as shown in Table 1.

P-FOL-IC3 [14] is concurrent work that extends FOL-IC3 by exploiting parallelism in formula search, introducing the invariant-friendly *k-Term Pseudo-DNF* to bound the search space, and randomly guessing some not-yet-inductive formulas to be eventually inductive, forcing their counterexamples to be excluded. P-FOL-IC3 has no theoretical guarantee and is less robust in practice due to its randomized nature; it failed in three out of five trials on stoppable Paxos, and two out of five trials on fast Paxos.

The tools discussed above, along with DuoAI, only verify safety properties of distributed protocols. Complementary work has explored connecting verification of protocols to their practical implementations [11, 31], and verifying liveness properties of distributed protocols [26].

AutoML [5, 18, 33] searches for machine learning models and hyperparameters, which may appear similar to finding inductive invariants. However, the inductiveness of invariants has strong correlation, which is difficult to capture for AutoML.

Many automated invariant inference tools have been built for other domains, mostly on learning loop invariants to verify sequential programs [3, 4, 6, 7, 15, 24, 25, 30, 32, 37, 39]. Invariant inference has been used to prove properties on inductive algebraic data types [16, 22], integer linear dynamical systems [19], and deep neural networks [2]. None of these methods consider nondeterminism in concurrent or distributed settings, thus they cannot be directly applied to distributed protocols.

10 Conclusions

DuoAI automatically and efficiently infers inductive invariants for verifying distributed protocols by reducing SMT costs. It introduces the minimum implication graph to define the structure of the invariant search space. This enables efficient enumeration of possible invariants, which are checked on samples from protocol simulation to reduce SMT queries. DuoAI guarantees that the enumerated candidate invariants are at least as strong as any correct invariants. DuoAI then runs top-down and bottom-up refinement in parallel. The former monotonically weakens the candidate invariants following the minimum implication graph. The latter divides the candidate invariants into an SMT-friendly universal inductive core and other noncore invariants, and searches for a small subset of noncore invariants that can be added to the core to prove the safety property of the protocol. Both top-down and bottom-up refinement have strong theoretical guarantees for finding inductive invariants, and their combination is effective at reducing SMT query costs for invariants with existential quantifiers. DuoAI dominates alternative tools in both the number of protocols it verifies and the speed at which it does so, including giving automated proofs for several Paxos variants.

11 Acknowledgments

Ji-Yong Shin provided helpful comments on earlier drafts. This work was supported in part by three Amazon Research Awards, a Guggenheim Fellowship, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080. Ronghui Gu is the Founder of and has an equity interest in CertiK.

References

[1] Decidability in Ivy. http://microsoft.github.io/ivy/decidability.html.

[2] Guy Amir, Michael Schapira, and Guy Katz. Towards scalable verification of deep reinforcement learning. In *Proceedings of the 21st Conference on Formal Methods in Computer Aided Design (FMCAD '21)*, pages 193–203, October 2021.

[3] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Solving constrained Horn clauses using syntax and data. In *Proceedings of the 18th Conference on Formal Methods in Computer Aided Design (FMCAD '18)*, pages 1–9, October 2018.

[4] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV '19)*, pages 259–277, July 2019.

[5] Matthias Feurer, Aaron Klein, Jost Eggensperger, Katharina Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Proceedings of the 29th Conference on Neural Information Processing Systems (NIPS '15)*, pages 2962–2970, December 2015.

[6] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Learning universally quantified invariants of linear data structures. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV '13)*, pages 813–829, July 2013.

[7] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, page 499–512, January 2016.

[8] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *Proceedings of the 13th NASA Formal Methods Symposium (NFM '21)*, pages 131–150, May 2021.

[9] Aman Goel and Karem A Sakallah. Towards an automatic proof of Lamport's Paxos. In *Proceedings of the 21st Conference on Formal Methods in Computer Aided Design (FMCAD '21)*, pages 112–122, October 2021.

[10] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*, pages 115–131, April 2021.

[11] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, pages 1–17, October 2015.

[12] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM*, 64(1), March 2017.

[13] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, page 703–717, September 2020.

[14] Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. Inferring invariants with quantifier alternations: Taming the search space explosion. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)*, pages 338–356, April 2022.

[15] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10)*, pages 328–343, November 2010.

[16] Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich. Beyond the elementary representations of program invariants over algebraic data types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, page 451–465, June 2021.

[17] Leslie Lamport. Byzantizing Paxos by refinement. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC '11)*, pages 211–224, September 2011.

[18] Trang T Le, Weixuan Fu, and Jason H Moore. Scaling tree-based automated machine learning to biomedical big data with a feature set selector. *Bioinformatics*, 36(1):250–256, January 2020.

[19] Engel Lefaucheux, Joël Ouaknine, David Purser, and James Worrell. Porous invariants. In *Proceedings of 33rd International Conference on Computer Aided Verification (CAV '21)*, pages 172–194, July 2021.

[20] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16), pages 357–370, January 2016.

[21] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, pages 370–384, October 2019.

[22] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. Data-driven inference of representation invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, pages 1–15, June 2020.

[23] Donald Monk. *Mathematical Logic*. Springer, October 1976.

[24] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE '17)*, pages 605–615, August 2017.

[25] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, page 42–56, June 2016.

[26] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *Proceedings of the ACM on Programming Languages*, 2(POPL), January 2018.

[27] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017.

[28] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, pages 614–630, June 2016.

[29] Oded Padon, James R Wilcox, Jason R Koenig, Kenneth L McMillan, and Alex Aiken. Induction duality: Primal-dual search for invariants. *Proceedings of the ACM on Programming Languages*, 6(POPL), January 2022.

[30] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: Learning loop invariants with continuous logic networks. In *Proceedings of 8th International Conference on Learning Representations (ICLR '20)*, March 2020.

[31] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, 2(POPL), January 2018.

[32] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. A data driven approach for algebraic loop invariants. In *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, pages 574–592, March 2013.

[33] Christian Steinruecken, Emma Smith, David Janz, James Lloyd, and Zoubin Ghahramani. The automatic statistician. In *Automated Machine Learning*, pages 161–173. Springer, May 2019.

[34] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, pages 662–677, June 2018.

[35] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, pages 357–368, June 2015.

[36] Doug Woos, James R Wilcox, Steve Anton, Zachary Tatlock, Michael D Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CCP '16)*, pages 154–165, January 2016.

[37] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, pages 106–120, June 2020.

[38] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-driven automated invariant learning for distributed protocols. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*, pages 405–421, July 2021.

[39] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, pages 707–721, June 2018.

Verifying Hardware Security Modules with Information-Preserving Refinement

Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

Abstract

Knox is a new framework that enables developers to build hardware security modules (HSMs) with high assurance through formal verification. The goal is to rule out all hardware bugs, software bugs, and timing side channels.

Knox's approach is to relate an implementation's wire-level behavior to a functional specification stated in terms of method calls and return values with a new definition called *information-preserving refinement (IPR)*. This definition captures the notion that the HSM implements its functional specification, and that it leaks no additional information through its wire-level behavior. The Knox framework provides support for writing specifications, importing HSM implementations written in Verilog and C code, and proving IPR using a combination of lightweight annotations and interactive proofs.

To evaluate the IPR definition and the Knox framework, we verified three simple HSMs, including an RFC 6238-compliant TOTP token. The TOTP token is written in 2950 lines of Verilog and 360 lines of C and assembly. Its behavior is captured in a succinct specification: aside from the definition of the TOTP algorithm, the spec is only 10 lines of code. In all three case studies, verification covers entire hardware and software stacks and rules out hardware/software bugs and timing side channels.

1 Introduction

A powerful approach for building secure computer systems is to factor out the core security functionality onto a separate device. For example, on the server side, certificate authorities use hardware security modules (HSMs) to store their signing key and sign certificates [10, 58]; credit card networks use HSMs for pin translation, secure re-encryption of payment requests during routing; cloud providers use HSMs to safeguard PIN-protected backup keys [9, 43, 47]; and some tax authorities require the use of an HSM to timestamp invoices. On the client side, the iPhone uses its secure enclave processor to enforce PIN guessing limits for unlocking the phone [15]; and users often rely on USB security keys to protect their authentication private key in the face of a compromised computer [65]. For simplicity, this paper refers to all of these types of devices as HSMs. These devices are in widespread use; e.g., there are hundreds of millions of deployed secure enclaves and security keys.

This approach defends against a broad class of attacks where an adversary gains access to any host computer that the HSM might be connected to, regardless of the specific attack vector (exploiting a buffer overflow, missing access control checks, or even gaining access to the administrator's SSH key). As long as the security of the overall system is rooted in the device, an adversary that controls the host cannot undermine the security of the overall system. Of course, the device must be correctly implemented to make sure that the adversary cannot compromise it, which in practice means that the device must provide simple, well-defined functionality.

Although HSMs are relatively simple, any vulnerability in their hardware or software can undermine their security. HSMs have suffered from bugs throughout the hardware/software stack, such as logic bugs, memory corruption, hardware bugs, and timing side channels [1–8, 21, 31, 45, 51, 68]. This paper presents an approach for ruling out such bugs through formal verification, with a particular focus on eliminating leakage through timing side channels.

Our approach is to relate the behavior of the HSM implementation at the wire level interface — the ground truth of what the host machine controls and observes at the digital level, which captures timing channels at a cycle-accurate level — to a functional specification of the methods that the HSM exposes. Figure 1 shows the implementation of a simplified PIN-protected backup HSM, which we use as a running example through the paper. The host connects to this HSM via two input wires and two output wires, which the host can read/write at every cycle. Figure 2 shows the functional specification for this HSM. It exposes two operations, `store` and `retrieve`. The specification does not have an operation for reading back the PIN, and it enforces a guess limit on PINs.

We relate a physical implementation to a functional specification with a new definition called *information-preserving refinement (IPR)*, inspired by definitions of zero-knowledge proofs in cryptography [40, 41]. IPR captures the notion that the implementation implements the spec, and that its wire-level I/O behavior leaks no additional information. In IPR, a *driver* describes the I/O protocol that a host computer can follow to get correct results from the HSM, describing how each spec-level operation translates to wire-level I/O with the HSM. The driver is a part of the specification (and is trusted). Its dual, an *emulator*, is a proof artifact that describes how wire-level behavior can be explained in terms of spec-level operations. The existence of an emulator shows that no matter what wire-level inputs are given to the device (including inputs that violate the I/O protocol), its outputs reveal no more information than the specification.

Applied to HSMs, IPR can capture subtle security bugs: for example, Figure 3 shows code that is correct and even crash safe but has a subtle bug involving persistence and

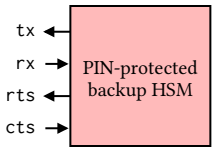


Figure 1: The physical implementation of the PIN-protected backup HSM. It connects to the host via 4 wires, speaking UART with flow control.

```
var bad_guesses = 0, secret = 0, pin = 0

def store(new_secret, new_pin):
  secret = new_secret
  pin = new_pin
  bad_guesses = 0

def retrieve(guess):
  if bad_guesses >= 10:
    return 'No more guesses'
  if guess == pin:
    bad_guesses = 0
    return secret
  bad_guesses = bad_guesses + 1
  return 'Incorrect PIN'
```

Figure 2: A functional specification for a PIN-protected backup HSM. The spec doesn't support reading out the PIN, and retrieval of the secret requires supplying the correct PIN. Limiting guesses prevents brute-forcing.

timing. The way this code gets compiled, the circuit takes longer to persist the incremented guess count, in the case of an incorrect guess, than it takes to zero the guess count, in the case of a correct guess (it takes longer to take the branch than to fall through). This can be abused to reset the guess count by repeatedly guessing every possible PIN and powering off the device after just enough cycles to reset the guess count in the case that the guess is correct (but not waiting long enough to persist if the guess is incorrect). Verifying IPR caught this bug in our implementation (§7.1.1). The buggy implementation doesn't enforce guess limits, which leaks more information than the specification, and this is prohibited by IPR.

Existing security definitions like noninterference or declassification either do not apply or are insufficient to capture the security of wire-level observations and arbitrary wire-level I/O as in the Knox setting (§9).

To be able to verify HSMs with IPR, we developed the Knox framework. Developers using Knox write HSM implementations using standard languages (i.e., Verilog and C code), write specifications in Knox DSLs, and use a combination of lightweight annotations and interactive proofs to show that the implementation is an information-preserving refinement of the specification.

To demonstrate that IPR and the Knox framework can be applied to HSMs and catch bugs in their implementations, we developed and verified three HSMs: a PIN-protected backup HSM, a password-hashing HSM, and an RFC 6238-compliant TOTP token [54]. The Knox HSMs do not have the implementation complexity of commercial HSMs: for example, the RISC-V processor they use is simpler than the ARM Cortex-M series embedded processors ubiquitous in security tokens such as SoloKeys. Still, the HSMs demonstrate many of the hardware and software complexities present in real HSMs. They all use an embedded processor (a RISC-V CPU) and interface with the host via digital I/O (UART), and the password hasher and TOTP token include hardware cryptographic accelerators. All three run application-specific C code, with some including cryptographic functionality, such as HMAC in the TOTP token. Knox proofs are end-to-end, encompassing hardware and software and showing that the implementation is free of exploitable hardware bugs, software bugs, and timing side channels.

```
// return error if PIN guess limit exceeded
// ...

// check PIN guess and update guess_count accordingly
if (!constant_time_cmp(&entry->pin, guess)) {
    entry->bad_guesses++;
    uart_write(ERR_BAD_PIN);
    return;
}
entry->bad_guesses = 0;

// output secret
// ...
```

Figure 3: Code snippet from an insecure `retrieve` implementation. `entry` points to persistent memory. The commit point depends on whether the PIN guess is correct.

In summary, this paper makes the following contributions:

- The definition of information-preserving refinement (IPR), which relates a physical implementation to a functional specification and captures that it: (1) implements the specification, and (2) leaks no additional information
- The Knox framework for proving that an HSM implementation satisfies its specification under the IPR definition
- An evaluation of the IPR definition and Knox's application to three simple HSMs

This paper applies IPR to HSMs, but we believe the definition is broadly applicable to other contexts for capturing non-leakage properties.

This paper has several limitations. The three HSMs verified using Knox are relatively simple: for example, they do not use public-key signatures, which are common HSM operations, because it is difficult to scale up proofs in Knox to handle sophisticated arithmetic needed for public-key implementations. Relatedly, for cryptographic operations such as public-key signatures, IPR requires the emulator to be efficient. Knox currently relies on a manual audit to ensure that the emulator does not brute-force secrets or run in exponential time (§8.1). Finally, IPR does not support true random number generators (TRNGs) — the functional specification has to be deterministic. We believe that a pseudo-random number generator is a reasonable workaround that fits into IPR (§8.2).

2 Threat model and security goal

This paper considers a powerful adversary that gains direct access to the wire-level digital I/O of the HSM, with the ability to set logic levels on the input wires and read logic levels on the output wires at every cycle. This captures many realistic attacks, such as an adversary that compromises the host computer and is able to send malformed commands or observe all wire-level outputs at every clock cycle. Such an adversary may be able to extract secrets from an HSM, even if that HSM operates correctly when the host computer is well-behaved.

Our threat model is focused on remote compromise of the host machine, one of the primary attacks that HSMs aim to defend against, so it does not include physical attacks on the HSM. While the adversary can perform arbitrary digital I/O to the HSM through a compromised host, remote compromise is unlikely to allow the adversary to violate the HSM's electrical specifications (e.g., supply 5V into an input wire expecting 3.3V logic or supply current to an output pin of the HSM) or observe analog characteristics of the I/O interface (e.g., measure analog voltage on a pin).

While the threat model includes (digital) timing side channels due to the level at which we model the host-HSM interface (wire-level I/O at every cycle), the threat model does not include arbitrary side channels [73] such as electromagnetic radiation [12], temperature [44], and power [49], because a remote attacker is unlikely to able to make such observations.

The goal of a Knox HSM is to be as secure as its specification. A host machine should be able to follow an I/O protocol to invoke spec-level operations on the HSM and obtain the correct outputs, but the host machine should not be able to abuse the wire-level interface to subvert the HSM and bypass its API or cause it to leak secrets.

3 Information-preserving refinement

The goal of information-preserving refinement (IPR) is to define what it means for an implementation with a wire-level physical interface to implement a functional specification and leak no additional information. IPR achieves this by establishing a bi-level correspondence between implementation and specification, at both the level of the functional interface (spec-level operations) and the physical interface (wire-level I/O). Illustrated in Figure 4, IPR is defined as an indistinguishability between two worlds: the real world, and an ideal world that is correct and secure by construction.

The real world models the host machine connected to the actual HSM implementation. The host can take a physical view of the device and directly perform arbitrary wire-level I/O (reading and writing the I/O pins at every cycle). The host can also take a functional view of the device and follow the HSM's I/O protocol, which is described by a *driver* that is part of the specification. The driver translates spec-level operations to wire-level I/O, describing how the host invokes

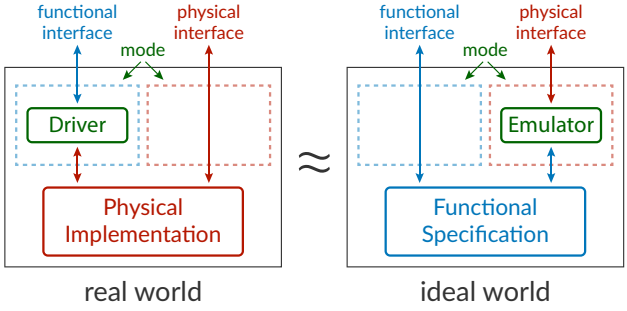


Figure 4: Information-preserving refinement (IPR), defined as an indistinguishability between a real world and an ideal world that is correct and secure by construction.

the operation and reads the return value by interacting with the HSM over its wire-level interface.

The ideal world is set up to provide the same interface as the real world but be correct and secure by construction. In the ideal world, a host machine that takes a functional view of the device invokes operations directly on the specification. To provide a physical view, an *emulator* mimics wire-level behavior, given only query access to the functional specification. The emulator is a dual of the driver; it translates wire-level I/O into spec-level operations. Unlike the driver, the emulator is merely a proof artifact. The ideal world can be instantiated with *any* emulator, and it remains secure by construction. IPR is defined to hold if there exists some emulator such that the real and ideal worlds are indistinguishable.

The host can switch between the functional view and the physical view at any time. Switching from the functional view to the physical view models compromise of the host machine; switching from the physical view back to the functional view models recovery (for example, by unplugging the device and moving it to an uncompromised machine). When switching views from physical to functional, in the real world, the driver is re-initialized; when switching from functional to physical, in the ideal world, the emulator is re-initialized.

The ideal world is correct and secure by construction. When the host takes a functional view of the device, operations are invoked directly on the specification, so the behavior is correct and secure by definition. Under the functional view, spec-level operations are not seen by the emulator. When the host takes a physical view of the device, the wire-level I/O behavior it observes is produced by an emulator that only has query access to the specification, so the physical interface leaks no more information than the specification exposes through its API. Furthermore, when the host switches back to the functional view of the device, it continues interacting with the same specification that was queried by the emulator, so the effect of any queries made by the emulator in order to mimic wire-level outputs is present in the specification state. In the ideal world, any execution, no matter how it switches between functional and physical interfaces, maps to some sequence of operations invoked on the specification.

Due to the indistinguishability that IPR requires between real and ideal worlds, any execution in the real world also maps to some sequence of operations invoked on the specification. In other words, when IPR holds, any attack that an adversary could execute on the real device could be transformed into an attack on the specification itself: the adversary could run the emulator and then execute the original attack using the emulator, which matches the implementation's wire-level behavior, given only query access to the specification. Indistinguishability between real and ideal worlds guarantees that the implementation is as secure as the specification.

Definition (Information-preserving refinement)**.** A physical implementation is an *information-preserving refinement* of a functional specification with respect to a driver if there exists an emulator such that the real world is indistinguishable from the ideal world as illustrated in Figure 4. ■

3.1 Applying IPR to HSMs

IPR, without explicitly talking about hardware, software, or timing side channels, captures exploitable bugs in all of those. If there were such exploitable bugs, IPR would not be satisfied: when there are implementation behaviors that can't be explained in terms of the specification, there does not exist an emulator that makes the real world and ideal world indistinguishable.

IPR relates any wire-level interaction with the HSM to an interaction with the specification. For example, suppose that the host machine follows the driver to perform a number of spec-level operations, and then it gets compromised, at which point it begins performing arbitrary I/O in an attempt to subvert the HSM. IPR, by requiring indistinguishability between the real and ideal worlds, says that this scenario corresponds to some sequence of spec-level operations, and that the arbitrary I/O reveals no more information than those spec-level operations do. Furthermore, IPR says that after the HSM is moved to an uncompromised host, normal operation can resume (as the host follows the driver), and that the behavior of the device will reflect any specification state changes that were a result of queries made by the emulator (any operations that were effectively invoked during arbitrary wire-level I/O).

The definition directly addresses host machine compromise by an adversary while the host is in between spec-level operations. It might seem like IPR only addresses arbitrary I/O that begins between these operations; however, a compromise in the middle of an operation can be thought of as a compromise that happens slightly earlier, at the start of the operation, and IPR covers this case.

Information-preserving refinement transfers both cryptographic and non-cryptographic security properties from the specification to the implementation. For example, the PIN-protected backup specification limits PIN guesses, and so IPR implies that the implementation enforces the guess limit as well. If it didn't limit guesses, it would reveal more information than the specification (through subsequent retrieve operations), which IPR prohibits. This rules out the subtle bug shown in Figure 3, even though the information disclosure manifests after the buggy code executes. If a specification computed signatures without revealing a key, then IPR would imply that the implementation also doesn't leak the key, including through its timing behavior.

4 Proving IPR

Knox models the specification and the implementation (§4.1) as state machines, relates the two with a refinement relation, and proves three properties: an initialization property (§4.2), functional equivalence (§4.3, indistinguishability of the functional view), and physical equivalence (§4.4, indistinguishability of the physical view), tying together these properties with the refinement relation. Together, these properties imply IPR.

4.1 Physical implementation

Knox models HSM implementations with a cycle-accurate description of their wire-level I/O behavior, covering hardware and software. Figure 1 shows the interface of a circuit implementing PIN-protected backup. The HSM interface allows for: (1) setting input wires, (2) reading output wires, and (3) waiting for the HSM to execute for a clock cycle of the HSM's internal clock.

In the case of the PIN-protected backup HSM, the UART `rx` and `cts` wires can be set and the `tx` and `rts` wires can be read at every cycle. The baud rate is independent of the HSM clock frequency; the IPR formalism itself has no notion of a serial port or baud rate, only wires and hardware-level clock cycles. The three main Knox case studies use UART, but simpler Knox examples use different I/O protocols.

The HSM model comprises the circuit state, a step function describing behavior for a single cycle, the initial state of the HSM (contents of non-volatile memory, such as ROM containing code and read-write persistent memory being zero-initialized), and a description of the power-on / reset behavior of the circuit (losing the contents of volatile memory).

4.2 Refinement relation and initialization

Knox uses a refinement relation R, a proof artifact supplied by the developer, to relate the state of the implementation to the state of the specification *in between spec-level operations*. That is, it is not required to hold at arbitrary steps of the circuit, only before/after spec-level operations, or after switching from the physical view to the functional view, which involves re-initializing the driver (which in our implementations, resets the circuit). Use of a common R connects functional equivalence and physical equivalence.

R relates states and usually includes an invariant that captures circuit quiescence (it holds in between spec-level operations). Figure 5 shows the refinement relation used in

$$\begin{aligned} \text{spec.bad\_guesses} = \text{swap32}(\text{impl.fram}[0..3]) \quad &\wedge \\ \text{spec.pin} = \text{impl.fram}[4..9] \quad &\wedge \\ \text{spec.secret} = \text{impl.fram}[10..19] \quad &\wedge \\ \text{Inv}(\text{impl}) \end{aligned}$$

Figure 5: A simplified version of the refinement relation used in the proof of PIN-protected backup. *impl.fram* refers to the persistent memory of the implementation. *swap32* performs a byte order swap. *Inv* is the invariant (not shown here).

```
(define (store secret pin)
  (send-byte #x02) ; command number
  (send-bytes pin)
  (send-bytes secret)
  (recv-byte)) ; wait for ack

(define (wait-until-clear-to-send)
  (while (get-output 'rts))
    (tick))) ; wait a cycle

(define (send-bit bit)
  (set-input 'rx bit)
  (for ([i (in-range BAUD-RATE)])
    (tick)))

(define (send-byte byte)
  (wait-until-clear-to-send)
  (send-bit #b0) ; send start bit
  ;; send data bits
  (for ([i (in-range 8)])
    (send-bit (extract-bit byte i)))
  (send-bit #b1)) ; send stop bit

(define (send-bytes bytes)
  (for ([byte bytes])
    (yield) ; wait for arbitrary number of cycles
    (send-byte byte)))
```

Figure 6: A code snippet from the PIN-protected backup driver. The function corresponding to a spec-level operation is shown in blue. Driver-language primitives are in red.

the proof of the PIN-protected backup HSM. It relates each variable in the specification to the persistent memory of the circuit.

Knox requires that the initial implementation state is related by R to the initial specification state.

4.3 Functional equivalence

Functional equivalence states that spec-level behavior is obtained from the implementation's wire-level interface by following the I/O protocol described by the driver. The driver is a program, written in Knox's driver language, that is part of the specification of the HSM. For every spec-level operation, the driver has a corresponding function that describes how the host invokes the operation on the HSM over its wire-level I/O interface.

For example, Figure 6 shows the driver for the PIN-protected backup HSM. The driver exposes a function corresponding to each spec-level function, such as `(store ...)`, implemented in terms of driver-language primitives for interacting with the implementation: `(set-input ...)` and `(get-output ...)` write the input wires and read the output wires, respectively; `(tick)` waits for the HSM to execute for a single cycle; `(yield)` models situations where the host is allowed to wait for an arbitrary number of cycles, e.g., in between sending bytes in an asynchronous protocol.

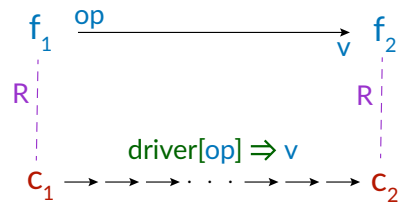


Figure 7: Functional equivalence: for all implementation states c_1 and spec states f_1 that are related by R, and for all spec-level operations op:
(1) the spec-level output v matches the driver output
(2) the final states c_2 and f_2 are related by R

Figure 7 defines functional equivalence: starting from circuit/spec states related by R, invoking an operation on the specification gives the same result as running the corresponding driver function against the circuit, and the final circuit/spec states continue to be related by R.

The HSM runs asynchronously from the host: its clock keeps ticking even if there is no operation to perform. To model this, the driver also describes a spec-level no-op: e.g., in the case of the PIN-protected backup HSM, the host sets the `rx` line high, indicating that it has nothing to transmit. Functional equivalence also covers this no-op case.

4.4 Physical equivalence

Physical equivalence states that wire-level behavior matching the real circuit's behavior can be obtained by running an emulator (with query access to the specification), capturing the notion that the circuit leaks no more information than the specification. The emulator in IPR is a dual of the driver: it is a program, written in Knox's emulator language, that implements wire-level interactions in terms of spec-level operations. Unlike the driver, the emulator is a proof artifact: if there exists an emulator that mimics circuit behavior, then physical equivalence holds.

An emulator exposes a function corresponding to each wire-level interaction: setting the input, getting the output, and running for a cycle. These are implemented in terms of emulator-language primitives for invoking spec-level operations (e.g., `(store ...)` and `(retrieve ...)`, for the PIN-protected backup). Besides the ability to make black-box queries to the functional specification, the emulator can maintain auxiliary state across emulating multiple cycles; the auxiliary state is initialized to a null value whenever the emulator is re-initialized.

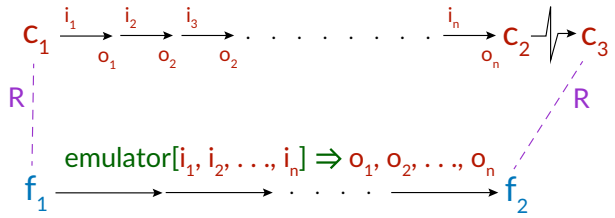


Figure 8: Physical equivalence: for all spec states f_1 and implementation states c_1 that are related by R, and for all wire-level inputs $i_1 \ldots i_n$:
(1) the circuit outputs $o_1 \ldots o_n$ match the emulator outputs
(2) the final states f_2 and c_3 (c_2 after a reset) are related by R

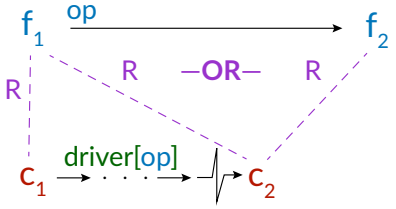


Figure 9: Crash safety: for all implementation states c_1 and spec states f_1 that are related by R, and for all spec-level operations op: if the driver is interrupted at any point, the post-reset state of the circuit c_2 is related by R to *either* f_1 or f_2.

Figure 8 defines physical equivalence: starting from circuit/spec states related by R, any wire-level I/O behavior exhibited by the circuit is matched by the emulator, which makes queries to the specification as it runs. Furthermore, the final specification state is related by R to the final circuit state (after the circuit is reset).

IPR is satisfied as long as there exists some emulator such that the real and ideal worlds are indistinguishable. Proofs of physical equivalence in Knox involve constructing an emulator (i.e., writing a program in the emulator language) that satisfies the definition of physical equivalence. Because the emulator is merely a proof artifact, the details of the construction do not matter, as long as the program satisfies the definition. The Knox case studies (§7) describe the techniques used in practice to write emulators.

4.5 Crash safety

Physical equivalence already covers the case of an interrupted spec-level operation, because an interrupted protocol-following execution can be viewed as a case of arbitrary I/O: physical equivalence guarantees that any wire-level I/O corresponds to *some* sequence of spec-level operations. However, we can state an additional property that is stronger: when the HSM is interrupted in the middle of an operation while the host is following the driver, the implementation is crash safe, acting either as if the operation never started or as if the operation completed successfully. Figure 9 defines this crash-safety property.

5 The Knox framework

The Knox framework uses hybrid symbolic execution [67] and SMT solvers to help developers prove IPR. Knox includes techniques to handle the challenges that arise when applying symbolic execution for proving functional equivalence and physical equivalence. In functional equivalence proofs, Knox handles the nondeterminism of `yield` in drivers by automatically finding fixed points (§5.1). In physical equivalence proofs, Knox supports reasoning about unbounded-length inputs using an approach we call *guided symbolic model checking* (§5.2). In both, Knox allows the proof developer to supply *hints*, untrusted guidance where the framework invokes the solver as necessary to ensure soundness (§5.3).

5.1 Nondeterminism

Knox verifies the functional equivalence property using symbolic execution of the driver-language program against the HSM implementation, comparing the execution of each driver operation against the corresponding spec operation. However, symbolic execution cannot directly handle the nondeterminism of `(yield)`, which has the semantics of the driver waiting for an arbitrary number of cycles while the HSM runs.

Knox addresses this by finding a fixed point of the circuit's step function at every yield point. During symbolic execution, the circuit's state is a symbolic term. Stepping the circuit produces a new symbolic term, and so on. At yield points, Knox computes a set of symbolic terms such that the set is closed under the circuit's step function, and it forks symbolic execution for each term in the set.

Closure is defined in terms of symbolic state subsumption. A symbolic term t under a path condition p, written as $t|p$, can be thought of as representing a set of concrete values, $[\![t|p]\!]$, the set of values that t can evaluate to for all possible assignments satisfying p of values to t's symbolic variables. A term t_1 under path condition p_1 is subsumed by a term t_2 under path condition p_2, written as $t_1|p_1 \subseteq t_2|p_2$, if $[\![t_1|p_1]\!] \subseteq [\![t_2|p_2]\!]$. For a set S of symbolic terms paired with path conditions, let $[\![S]\!] = \{[\![t|p]\!] : t|p \in S\}$. Finally, call S a fixed point of the step function if $\forall x \in [\![S]\!], \text{step}(x) \in [\![S]\!]$.

Knox includes an efficient algorithm for subsumption checks, and fixed points are found through iteratively calling the step function on the symbolic circuit state to build up a set of symbolic terms. Once a fixed-point S is found, symbolic execution proceeds for each of the $t|p \in S$, similar to how branching produces multiple paths to be checked.

Left unchecked, multiple `(yield)`s can result in an exponential number of cases to check, analogous to the problem of branching resulting in path explosion in symbolic execution. For this reason, Knox uses untrusted `(merge)` hints in the driver at points where some branches could be merged together. At merge points, Knox uses subsumption checks to automatically find a smaller set of symbolic terms $|S'| \leq |S|$ that still represent all the concrete values included in the original, i.e., $[\![S]\!] \subseteq [\![S']\!]$, which addresses case explosion.

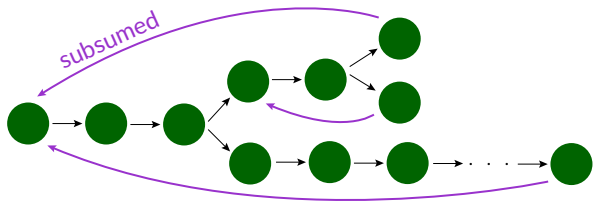


Figure 10: An illustration of *guided symbolic model checking* exploring a state space. Each green circle is a symbolic term representing a set of states. Black arrows show STEP invocations and purple arrows show SUBSUMED invocations.

5.2 Unbounded-length inputs

In Knox, emulators can be symbolically executed with black-box query access to a functional specification. Unlike the functional equivalence property which considers a single (spec-level) input, the physical equivalence property considers an arbitrary-length sequence of (wire-level) inputs, so Knox can't prove the physical equivalence property in the same way. Symbolic execution could verify this property for a fixed-length input, but it cannot directly handle arbitrary-length input.

The standard approach to handling arbitrary-length inputs is to write down an inductive invariant and reason about one step at a time. This approach does not work for large circuits because of the infeasibility of manually writing down the inductive invariant. It would have to include an invariant of circuit execution, capturing which states are reachable and which are not, and it is infeasible to manually write down exactly how CPU microarchitectural registers, peripheral registers, RAM state, etc. are related to each other at every cycle of execution of the software.

Instead, Knox uses an approach that we describe as *guided symbolic model checking*. At a high level, Knox uses a model-checking-style approach to start from the initial states of the circuit and emulator in the definition of physical equivalence, explore all reachable states, and ensure that the circuit's behavior matches the emulator's behavior and the recovery condition holds at every step. Exploration starts out at a circuit state c_1, an emulator state e_0 (the initial emulator auxiliary state, null), and functional spec state f_1, where both f_1 and c_1 are symbolic terms, and R is assumed to relate f_1 and c_1. Knox can step the circuit and step the emulator, given the same symbolic input, and check that their outputs match. Knox repeats this process until it has explored all reachable states.

This model-checking process involves guidance from the developer in the form of a proof script. Knox provides two primitives that allow the developer to guide exploration of the state space:

- STEP steps the circuit and the emulator/spec (with the same symbolic input) for one cycle and verifies the output equivalence and recovery properties for that single cycle
- SUBSUMED checks that the state currently under consideration is subsumed by a state that was explored earlier, "tying the knot" and finishing a branch of the exploration

Figure 10 illustrates how STEP and SUBSUMED let the developer guide the model checker to explore the state space. In addition to these primitives, the developer uses additional *hints* (§5.3) to safely manipulate symbolic terms and help the model checker efficiently explore the state space.

An alternative view of this process is that it incrementally builds up the induction hypothesis that would have been used in an induction-based approach. Once model checking has explored all reachable states, it has visited a set of states S that includes the initial circuit/emulator/spec state where R holds, and the set S has the property of being closed under the circuit/emulator step functions, and the property of matching outputs for a single cycle holds for every state in S. The induction hypothesis is that the state is contained in S.

The proof script is untrusted, and Knox checks that the state space is fully explored. At worst, an incorrect proof script can result in poor performance or Knox reporting that the state space has not been fully explored.

5.3 Hints

In both functional equivalence proofs and physical equivalence proofs, relying only on hybrid symbolic execution quickly results in an explosion in term size, and in the case of HSMs involving cryptography, queries that make the SMT solver time out.

Knox addresses this with untrusted (solver-checked) human guidance called *hints*. Knox has 8 primitive hints:

- CASE-SPLIT performs case analysis
- CONCRETIZE invokes the solver to prove that a symbolic term is concrete and replaces it with the concrete value
- OVERAPPROXIMATE replaces a term with a fresh variable
- WEAKEN weakens the current path condition
- REPLACE rewrites or simplifies terms
- REMEMBER, SUBSTITUTE, and CLEAR effectively allow marking terms as opaque to symbolic execution and substituting in their values later

Furthermore, Knox supports writing higher-level *tactics* that can reflect on the current state of symbolic execution and invoke primitive hints (or other tactics). A tactic might, for example, analyze the state of the circuit to determine if a CPU is about to branch, and in that situation, it can invoke a CASE-SPLIT hint with the appropriate cases constructed based on analyzing the symbolic circuit state.

All invocations of hints are verified by the Knox framework with an call to the SMT solver when necessary. Hints are untrusted: at worst, hints can be incorrect and fail (e.g., when attempting to replace a term with an unequal term), which will result in an error message to the user, or the given hints can be inadequate to ensure good performance, in which case verification will be slow or fail to terminate.

6 Implementation

The Knox framework builds on top of Racket [37] and the Rosette solver-aided programming language [67], and it relies on the Z3 SMT solver [32]. To compile circuits to a shallow embedding in Rosette, Knox uses GCC and its RISC-V backend to compile C code, Yosys [69] and its SMT-LIB backend to process Verilog, and `#lang yosys` (700 LOC of Racket and Rosette) from Notary [16] to convert the SMT-LIB output into a Rosette model.

The Knox framework's core — the semantics for the driver and emulator languages, and the tools for verifying functional equivalence and physical equivalence — is implemented in 3000 lines of Racket and Rosette. Achieving good verification performance required many optimizations and some new techniques, including symbolic state serialization, term substitution, fixpoint finding, state merging, and a new algorithm for symbolic subsumption checking based on a disjoint-set data structure.

The case studies are implemented in Verilog, C, and assembly (summarized in Figure 16). The case studies run on a $65 1BitSquared iCEBreaker development board, which has a Lattice iCE40UP5K FPGA, and use an open-source FPGA toolchain: the Yosys synthesis tool, the nextpnr place and route tool [72], and Project IceStorm [70] to create the bitstream and flash the FPGA. The FPGA connects using an FTDI cable to a host computer running Linux, for which we wrote client libraries for the three HSMs.

Figure 11 shows an overview of the different components that the developer writes when using Knox to verify an HSM. The functional specification, physical implementation, and refinement relation R are common inputs, used when verifying both functional equivalence and physical equivalence. When verifying functional equivalence, Knox takes as additional input the driver, along with hints to guide symbolic execution. When verifying physical equivalence, Knox takes as additional input the emulator and a proof script. The functional specification and the driver, highlighted in green, comprise the code written by the developer that is trusted. Other components, the HSM implementation and proof artifacts, are verified by the framework. Similar to other tools based on symbolic execution, when verification in Knox fails, the framework can provide a concrete counterexample, aiding the developer in debugging the implementation or the proof.

Source code for Knox and the case studies is available at https://github.com/anishathalye/knox.

7 Evaluation

To evaluate information-preserving refinement and the Knox framework, we ask the following questions:

- Can IPR and Knox be applied to HSM hardware/software?
- What types of bugs does verification prevent?
- What is the performance of the Knox framework?
- What is the performance of HSMs verified with Knox?

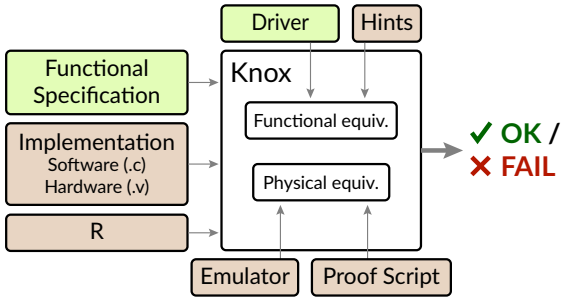


Figure 11: An overview of the Knox workflow. Trusted inputs are shown in green.

Methodology. We evaluate the first two questions through case studies (§7.1) that formally verify three HSMs with different types of specification and implementation complexity: a PIN-protected backup HSM, a password-hashing HSM, and an RFC 6238-compliant TOTP token [54]. To answer questions related to verification performance and the performance of the HSM implementations, we report on measurements (§7.2).

7.1 Case studies

7.1.1 PIN-protected backup HSM

Specification. A simplified PIN-protected backup HSM (Figure 2) was a running example through this paper; we verified an HSM with additional functionality: storing multiple secrets, each protected by its own PIN, and indexed by a slot number. The specification exposes four functions: `status`, `store`, `retrieve`, and `delete`. The specification demonstrates support for non-cryptographic security properties, such as the guess limit on PINs.

Implementation. Figure 12 shows a schematic of the implementation. It uses the PicoRV32 RISC-V CPU and the SimpleUART peripheral from the PicoSoC [71] with minimal modifications: we removed asynchronous reset from the CPU and added hardware flow control to the UART. The HSM uses ferroelectric RAM (FRAM) for persistent storage. Knox requires cycle-accurate models of the entire hardware, and FRAM has simple cycle-precise behavior, supporting durable word-level writes in a single cycle. For convenience, to avoid wiring an external chip, the prototype uses FPGA Block RAM in place of FRAM for the experiments. In total, the HSM hardware is described in 2670 lines of Verilog.

The software is written in a combination of C and assembly. To simplify verification, the HSM uses a strategy inspired by Notary [16] to minimize variation in the states that the hardware can be in. The HSM uses a reset-based design: the SoC is held in an "embryo" state until the host is ready to perform an operation, and after the HSM performs a single operation, it enters the embryo state again until the host begins the next operation. This is done through a combination of hardware and software: the HSM's `cts` input doubles as a signal that the host is ready to perform an operation, and a

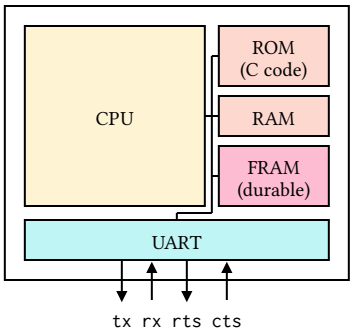


Figure 12: A schematic of the PIN-protected backup HSM.

small amount of logic implemented in hardware holds the rest of the circuit in a reset state until the host is ready to perform an operation. After the HSM performs a single operation, it signals this power management hardware to reset the SoC and return to the embryo state.

The software on the HSM includes a driver for the memory-mapped UART peripheral, along with code to implement each of the operations. The `main` function of the HSM reads a command and arguments from the UART, calls the appropriate handler, and then shuts down. The code for the HSM is written in 150 lines of C and 40 lines of assembly.

Verification and bugs caught. Knox physical equivalence proofs are constructive. We designed the emulator for the PIN-protected backup as follows. The emulator runs a copy of the circuit with dummy data. The emulator does not have access to the data in the real circuit, in particular the read-write persistent memory, but the structure of the circuit and the code in the ROM is common knowledge. The emulator carefully watches the internal state of its copy of the circuit: when the circuit is about to perform an operation, the emulator reads input data out of its circuit's state and translates it into a spec-level input, makes a query to the specification, and injects the result back into its circuit's state, so that the output behavior of the circuit copy matches that of the real circuit. For example, for the retrieve operation, when the emulator sees that its circuit copy has just completed the equality comparison, the emulator extracts the slot number and PIN guess from the circuit copy's RAM, makes a `retrieve` query to the specification, and injects the result (match or no match) back into its copy of the circuit, also injecting the secret into the appropriate location in memory if the guess indeed matched. All of the emulators for the case studies follow this general construction. Through the physical equivalence proof, we show that all implementation-level behavior can be explained with spec-level behavior, proving that the implementation leaks no more than the spec.

Verifying physical equivalence catches classic security bugs, such as a bug where the implementation's timing behavior leaks how many bytes of the PIN guess matches due to using `strcmp`. This information is not revealed by the spec — which only reveals whether or not the guess is correct (and the secret, when the guess is correct), not how many bytes of the guess match the PIN — so IPR prevents the implementation from leaking it. For such a buggy implementation, an emulator satisfying the IPR definition doesn't exist: the emulator doesn't have direct access to the true PIN (only query access through the specification), so its behavior can't match the real circuit's leaky behavior.

```
var secret = 0

def config(new_secret):
  secret = new_secret

def hash(password):
  return sha256(password || secret)
```

Figure 13: The functional specification for the password-hashing HSM.

Verifying physical equivalence caught a subtle security bug involving persistence and timing. Figure 3 shows a code snippet from the insecure implementation. The compiler happens to compile this code using a branch instruction, branching in the case where the guess is incorrect, and the CPU implementation takes longer to take the branch than to fall through to the next instruction. This has the effect that the circuit takes longer to write the updated `bad_guesses` value to persistent memory in the case of an incorrect PIN guess (where it's executing `entry->bad_guesses++`) than in the case of a correct PIN guess (where it's executing `entry->bad_guesses = 0`). An attacker can abuse this to reset the guess count by guessing a PIN, powering off the device after just enough cycles to reset the guess count in the case that the guess is correct (but not waiting long enough that `entry->bad_guesses++` has a chance to run, in the case that the guess is incorrect), and repeating this process for every possible PIN. This is not a correctness or even a crash-safety bug: this insecure implementation is both correct and crash safe. However, physical equivalence prevents this security bug in the implementation. The bug is fixed by using constant-time code to make the commit point of the operation independent of whether the PIN guess is correct.

7.1.2 Password-hashing HSM

Specification. Figure 13 outlines the specification for the password-hashing HSM. It includes a specification of SHA256 (not shown) that follows FIPS 180-4 [57]. The password hasher is configured with a secret, and then it computes salted hashes using the stored secret. There is no function to retrieve the secret after it is stored. The specification also guarantees that future operations cannot leak past inputs, because the secret cannot be read back, and passwords are not stored.

Implementation. The hardware is similar to that of the PIN-protected backup HSM. This HSM adds a hardware SHA256 cryptographic accelerator (about 300 lines of Verilog), which

implements the SHA256 block function. We originally used an off-the-shelf SHA256 core [66], but we switched to a custom implementation to minimize FPGA area so the design would fit on our low-cost board. The software includes a driver for the SHA256 peripheral, which implements the message padding function and drives the memory-mapped SHA256 peripheral, one block at a time. The software is written in 200 lines of C code and 40 lines of assembly.

The HSM needs to be crash safe, and the specification has operations that require updating multi-word values (the secret, in our specification, is 20 bytes), but the FRAM only supports word-level (32-bit) atomic writes. For this reason, the HSM uses a simple journaling strategy where it keeps two copies of the state in persistent memory and uses a flag to determine which one is active. To do an atomic write, the HSM writes to the inactive region and then toggles the flag.

Verification and bugs caught. The functional equivalence proof caught a bug in the hardware implementation, in the integration of the SHA256 peripheral into the SoC. The memory-mapped SHA256 peripheral's chip select input was being set based on the CPU's `mem_addr` bus, but the `mem_addr` is uninitialized on reset (it becomes stable after a couple cycles), so the SHA256 hardware could receive an unintended command right at boot. This bug would be difficult to catch through testing because it is only triggered in rare cases, when the uninitialized address bus contains a particular value on reset. The bug was fixed by adding an additional condition that `mem_valid` was also asserted (which all the other peripherals did, but the SHA256 peripheral didn't when it was first integrated).

Verification caught a security bug in the software, where the code branched on the flag indicating which region of persistent memory was active, and so there was observable timing variation where the circuit leaked more than the spec. Leaking which region of memory was active effectively leaked the parity of the total number of `hash` operations that the HSM had processed, which the specification does not expose. We fixed this by writing more careful C code that GCC compiled without branches so that there was no leakage.

7.1.3 TOTP token

Specification. Figure 14 outlines the specification for the TOTP token. It includes a specification of the TOTP algorithm (not shown) that follows RFC 6238 [54], which relies on HOTP [53], HMAC [46] and SHA1 [57]. The spec doesn't support reading back the secret after it has been set. It allows computing TOTP values given a timestamp supplied by the host machine, but it doesn't allow rewinding the timestamp. It supports an `audit` function to get the last timestamp value, to be able to identify if the HSM was ever abused to compute future TOTP values.

Implementation. The hardware is similar to that of the password-hashing HSM, except this token uses a hardware

```
var secret = 0, last_timestamp = 0

def set_secret(new_secret):
  secret = new_secret

def get_totp(timestamp):
  if timestamp < last_timestamp:
    return 'Cannot rewind timestamp'
  last_timestamp = timestamp
  return totp(secret, timestamp)

def audit():
  return last_timestamp
```

Figure 14: The functional specification for the TOTP token.

```
/* old implementation:
uint32_t s = (buf[offset]   & 0x7f) << 24
           | (buf[offset+1] & 0xff) << 16
           | (buf[offset+2] & 0xff) <<  8
           | (buf[offset+3] & 0xff);
*/
uint32_t s = 0;
for (int i = 0; i < 0x10; i++) {
    uint32_t match = ((i != offset) - 1);
    s += ((buf[i]   & 0x7f) & match) << 24;
    s += ((buf[i+1] & 0xff) & match) << 16;
    s += ((buf[i+2] & 0xff) & match) <<  8;
    s += ((buf[i+3] & 0xff) & match);
}
```

Figure 15: Rewriting TOTP dynamic truncation to avoid symbolic memory addresses.

SHA1 cryptographic accelerator. Its software includes a driver for the SHA1 peripheral that implements message padding, along with a software implementation of HMAC and the TOTP algorithm. Part of the TOTP algorithm is implemented in assembly, carefully written to prevent timing side channels. In one situation, we had to modify C code to be more amenable to symbolic execution, avoiding symbolic memory addresses in favor of fixed addresses and bit-twiddling tricks, as shown in Figure 15. The software for the TOTP token comprises 300 lines of C code and 60 lines of assembly.

Verification and bugs caught. The TOTP token uses a strategy matching the password hasher for achieving atomic state updates. Verifying functional equivalence caught a crash-safety bug where a struct field was missing a `volatile` qualifier and the compiler re-ordered a commit point (toggling the flag) before a write that should happen first (updating state in the inactive region of memory).

The emulator for the TOTP token follows the same basic construction as the others. One interesting detail: the `get_totp` implementation branches based on whether the timestamp is less than the last seen timestamp value; because the timestamp is a 64-bit value and PicoRV32 uses a 32-bit architecture, this turns into a number of comparisons/branches. The emulator, in order to make sure its behavior matches the real circuit's timing behavior, calls the `audit` function to retrieve the real last timestamp value and inject it in place of the

HSM	Spec		Driver	HW	SW	Proof
	core	total				
PIN backup	32	60	110	2670	190	470
PW hasher	5	150	90	3020	240	650
TOTP	10	180	80	2950	360	830

Figure 16: Lines of code for case studies. Lines of code for the spec are broken down into "core" and "total", where core is the main HSM functionality and doesn't include boilerplate or definitions of functions like SHA1, HMAC, and TOTP.

dummy data in the circuit copy, so that the timing behavior matches the real circuit. Also, the commit point for the TOTP operation is right after saving the new timestamp value, before the actual call to the TOTP function, so the emulator calls the functional specification's `get_totp` operation at the commit point in order to satisfy the recovery condition, stashes the output in auxiliary state, and when the circuit copy gets to the point where it's returning from its call to the TOTP function (computing with dummy data), injects the cached return value in place of the dummy value in the circuit.

The physical equivalence proof caught an issue with the TOTP implementation: it was using the C modulus (%) operator to compute the final $\mod 10^6$ operation, but this operation had variable latency dependent on its input, which leaks information that is not available in the functional specification. The spec doesn't reveal the output of the HMAC or dynamic truncation, only the final 6-digit code. The fix was to implement this functionality in constant time, which we did in assembly code using `sltu` and bitwise/arithmetic instructions.

7.1.4 Summary

IPR and Knox can be applied to simple HSM hardware and software. A design goal of the Knox HSMs, the implementations are minimal, using simple hardware throughout the SoC (e.g., a small RISC-V processor, simpler than the ARM Cortex-M found in many security tokens). Still, the Knox HSMs have implementation features found in real-world HSM hardware (e.g., microprocessor, I/O peripheral, persistent memory, cryptographic accelerator) and software (e.g., peripheral drivers, cryptography, crash safety), and Knox verification covers all of these.

Knox specs are succinct and proofs are manageable. Figure 16 shows lines of code in the spec and driver, implementation (hardware and software), and proof required for verifying each HSM. We break down spec lines of code into "core" and "total", where core doesn't include boilerplate or definitions of functions like SHA1, HMAC, and TOTP. Knox specifications are as short as their pseudocode: for example, aside from the definition of the TOTP algorithm as specified in RFC 6238, the core of the TOTP token specification in Knox is only 10 lines of code, as shown in Figure 17.

```
(struct state (secret last-ts))

(define s0 (state (bv 0 160) (bv 0 64)))

(define ((set-secret secret) s)
  (result #t (state secret (state-last-ts s))))

(define ((get-otp ts) s)
  (if (bvult ts (state-last-ts s)) (result (bv 0 32) s)
      (result (totp (state-secret s) ts)
              (state (state-secret s) ts))))

(define ((audit) s)
  (result (state-last-ts s) s))
```

Figure 17: The core of the Knox specification for the TOTP token. The definition of `totp`, not show here, is a pure function that follows the spec in RFC 6238.

HSM	FE-N	FE-N+C	FE	FE+C	PE
PIN backup	1	10	**209**	962	**8**
PW hasher	1	6	**74**	238	**4**
TOTP	3	8	**44**	141	**8**

Figure 18: Time taken (in minutes) for verification by Knox. FE is functional equivalence; the -N variation disables nondeterminism in the driver; +C adds verification of crash safety. PE is physical equivalence. The two bolded columns, FE and PE, together imply IPR.

Knox catches bugs throughout hardware/software. Verification caught bugs across hardware (e.g., SHA256 peripheral initialization, in the password hasher) and software (e.g., compiler re-ordering a commit point, in the TOTP token), including timing side channels (e.g., variable-time modulus, in the TOTP token) and subtle bugs involving hardware, software, timing, and persistence (e.g., commit point dependent on the PIN guess being correct, in the PIN-protected backup).

7.2 Performance

Verification performance. Figure 18 shows Knox's verification performance, evaluated on a 2014-era Intel i7-5930K. The implementation is currently single-threaded. Most of the time in functional equivalence proofs is due to nondeterminism (yield and merge) or verifying crash safety. The relatively low performance of verifying PIN-protected backup is due to performing case analysis on the slot number, which causes many paths to be explored independently.

When developing functional equivalence proofs, we usually begin by disabling driver nondeterminism and verification of crash safety. This significantly reduces verification time, and the tighter feedback loop speeds up the initial proof development process. After verification completes successfully in this simplified setting, we add back complexity and fix up the implementation and proof as needed.

Implementation performance. The case studies showed that hardware or software may need to be modified to satisfy

Metric	Baseline	Verified	
FPGA LUTs	3966	3962	(−0%)
Max clock freq	20.01 MHz	20.53 MHz	(+3%)
Code size	2412 B	2592 B	(+7%)
TOTP op latency	0.73 ms	0.83 ms	(+14%)

Figure 19: Overhead of modifications to TOTP token.

the strict definition of IPR and simplify verification. The TOTP token required the most modifications among the case studies. Figure 19 shows the impact of these modifications on hardware (FPGA area and maximum clock frequency) and software (code size and performance). Code performance was measured at a clock of 12 MHz and baud rate of 2M for the `totp` operation. Most of the slowdown results from the modification to the dynamic truncation code (Figure 15).

The verified TOTP token can perform TOTP operations with a latency of 0.83 ms, which is fast enough for interactive use [50]. The other HSMs have similar per-operation latency.

8 Discussion

This section elaborates on some of the design decisions made in IPR and Knox and discusses their implications.

8.1 Emulator efficiency

To meaningfully apply IPR to specifications that involve cryptography, the adversary must be efficient, and therefore, the emulator must be efficient as well. Without an efficiency requirement, an implementation that, for example, leaks an RSA signing key, could be justified by an emulator that calls the specification to get the public key, factors products of large primes in exponential time to compute the private key, and then perfectly mimics the physical interface because it has determined the implementation's internal state.

The emulator must satisfy a coarse-grained notion of efficiency: being prohibited from performing exponential-time computation and brute-forcing secrets. Without an efficiency requirement, information-preserving refinement captures an information-theoretic notion of information preservation, rather than a computational one.

The Knox framework does not fully formalize or mechanically verify emulator efficiency. Instead, the proofs rely on a manual audit of the emulator code. The emulators we construct are simple, so the efficiency property is easy to check. In fact, the Knox emulators in our case studies satisfy a stricter definition of efficiency than necessary — per cycle of the circuit that they emulate, they perform at most one query to the specification and perform computation roughly equivalent to what the circuit does in one cycle — meaning that an adversary could run the emulator with computational resources equivalent to the circuit itself.

8.2 Randomness

Functional specifications in IPR are deterministic, so IPR cannot be used to verify HSMs that use true random number generators (TRNGs). As an alternative, HSMs can use cryptographically-secure pseudo-random number generators (CSPRNGs), and this fits into IPR, because IPR supports internal state. The specification can internally use a CSPRNG, the spec can be augmented to expose an operation to add entropy to the CSPRNG, and this operation can be called by the host at device initialization time (and again at any time later) to seed the random number generator. IPR ensures that the CSPRNG's internal state cannot be leaked by the implementation.

8.3 Allowed leakage

IPR enforces that the implementation leaks no more than the specification; sometimes, it is desirable to allow the implementation to leak some non-sensitive information, e.g., the current `bad_guesses` count in the PIN-protected backup. This fits in to IPR: the leakage can be expressed as a spec-level `leak` operation. Knox supports leakage specifications using this strategy, and it allows the user to skip proving functional equivalence for `leak` operations (a well-behaved host does not need to invoke this operation; it is only relevant for modeling leakage as part of physical equivalence).

8.4 Monolithic end-to-end verification

Knox performs monolithic end-to-end verification, which has some benefits over modular verification. There is no need to define intermediate specifications and prove that layers satisfy these specs; there is no distinction between hardware and software, or a notion of, e.g., an instruction set architecture. Knox simply reasons about the cycle-accurate behavior of the entire circuit. If the circuit happens to contain a CPU that runs some software, the software is "inlined" into hardware (the initial contents of a ROM, for example).

Knox uses symbolic execution, and due to performing symbolic execution end-to-end across software and hardware, symbolic execution can be kept as concrete as possible, which improves performance. For example, Knox doesn't attempt to prove a CPU correct (that it executes *any* program correctly); this would require reasoning about symbolic instructions/programs. Instead, a proof of a Knox HSM only shows (indirectly) that the CPU executes the HSM's particular software correctly, which is an easier task.

Lack of modularity could be a challenge when scaling up Knox to more sophisticated HSM implementations, because end-to-end symbolic execution across hardware and software will perform poorly as hardware gets more sophisticated, and the proof developer might have trouble with non-modular reasoning as complexity increases. But modular reasoning about security properties is also challenging: e.g., proving software correct with respect to an ISA specification is inadequate for proving absence of timing side channels.

A potential approach to this problem could involve structuring the HSM implementation to delay responses until a worst-case execution time bound, and then specializing the verification framework to reason about HSMs following such a design. With such a structure, precise Knox-style reasoning about software executing on hardware may not be necessary, making it possible to separately reason about correctness (following standard approaches) and then reason about worst-case execution time bounds of software executing on hardware to show a top-level property like IPR.

9 Related work

Noninterference. Noninterference [39] captures confidentiality properties in systems where high-sensitivity inputs should not affect low-sensitivity outputs, which are separate from high-sensitivity outputs. seL4 [55], mCertiKOS [30], Komodo [36], and Nickel [64] verify noninterference properties such as process isolation. HACL* [62, 74], Vale [25, 38], EverCrypt [63], and Jasmin [14] phrase freedom from timing side channels in crypto code as noninterference by defining a leakage trace (the low-sensitivity output) that captures adversary observations, such as every program counter value, and showing that two executions that have matching public inputs but differing secrets (high-sensitivity inputs) produce identical leakage traces.

The Knox setting does not have separate low/high-sensitivity outputs: there is just one output, the logic levels on the output wires at every cycle, and this is what the adversary observes. Noninterference does not hold in the Knox setting: the output can and will be secret-dependent. Instead, IPR says that the output does not leak more information than the spec, which is not a noninterference property.

Declassification. Noninterference with declassification [56] separates low and high-sensitivity inputs (i.e., public and secret inputs) and supports controlled influence of secrets on outputs through an explicit declassify function that marks secret-dependent values as safe to output. Ironclad [42] uses this style of security definition; the proofs cover only software, not hardware, and do not rule out timing side channels.

The Knox setting does not separate low and high-sensitivity inputs. There is just one input, the logic levels on the input wires at every cycle. IPR says that after the HSM receives inputs from the driver (i.e., corresponding to a spec-level operation), its future behavior does not leak more information than the specification, which is not a declassification property.

Ironclad contains a PassHash app similar to the Knox password hasher. PassHash generates a secret internally, from a computer's TPM, and it services network requests: given a password, it returns a hash of the password salted with the secret. The secret is a high-sensitivity input (from the TPM) and the password is low-sensitivity input (from the network); the security definition is phrased as noninterference with declassification, allowing the final output to the network to depend on the secret in a controlled way. In the Knox HSM, the secret is not generated internally but received from the host, and both secrets and passwords are received over the same input wires (there are no separate public and secret inputs). A declassification-style definition does not apply. IPR says that the implementation's behavior can't leak more information than the specification, so for example, after a host sets the secret, the HSM can't leak the secret. IPR also gives the same property with passwords: the HSM can't leak passwords that were input earlier. In contrast, the noninterference property for PassHash doesn't prevent the implementation from leaking passwords, because they are low-sensitivity inputs.

Hardware/software verification. A long line of work performs end-to-end verification of functional correctness properties for hardware/software systems, with an emphasis on modular verification [13, 23, 35, 48]. Proving functional correctness does not rule out timing side channels, while addressing side channels is a central goal of IPR and Knox.

Knox uses Notary's toolchain to convert C/Verilog to Rosette models, and Knox uses Notary's idea of reset-based design for simplifying verification [16, 52]. Knox solves a different problem than Notary. IPR is a new security definition for HSMs that captures the notion that a hardware/software implementation satisfies a functional specification and leaks nothing more, and Knox is a framework for proving this property, including support for writing specifications, encoding drivers and emulators, and proving correctness and security. Notary's focus is a hardware/software architecture for better isolation between multiple mutually-distrustful agents running on the same device, and Notary only verifies a simple (but key) property of an embedded system and its boot code, that all internal state is cleared after reset.

Simulation-based definitions of security. IPR is inspired by simulation-based definitions of security for multiparty computation (MPC) and universal composability (UC) [28, 40, 41]. The Knox emulator is similar to the MPC simulator, which formalizes the notion of zero knowledge in an MPC protocol. Knox uses this concept to define non-leakage for a hardware/software system.

Verified cryptography. Some tools [18, 19, 24, 61] verify cryptographic properties of functional specifications and protocols. These are complementary to Knox, as illustrated by work that formally analyzes HSM interfaces [26, 33].

Other works prove functional correctness of crypto implementations [17, 22, 27, 29, 34]. Some of these provide side-channel resistance with cryptographic constant-time code, which can be compiled to machine code while preserving constant-time [20], but the security property does not go down to the hardware / wire I/O level.

Verifying efficiency. Cryptographic proofs generally reason about efficient adversaries, and some frameworks for verified cryptography support proving polynomial bounds on

programs' running time [18, 61]. Knox could follow their approach, adding a cost semantics for the emulator language, to formally reason about emulator efficiency.

Secure compilation. In Knox, the circuit is not derivable from the specification via compilation, but the IPR definition bears some resemblance to the properties secure compilers guarantee about their compilation results. Fully-abstract compilers [11] preserve and reflect observational equivalence from the source to the target language. Some security properties can be stated as program equivalences [60], but IPR's non-leakage property is not captured by this type of definition. In fact, some Knox specifications such as the password hasher have no instances that are observationally (extensionally) equivalent but not intensionally equal, so a secure-compilation-style equivalence preservation at the circuit level would be vacuous. Trace-preserving compilation [59] preserves trace equivalence between source and target and handles invalid target-level inputs. The definition is not general enough to apply to the HSM setting because source-level inputs don't map to single target-level inputs (function call to wire input for a *single* cycle), and there is no notion of "ignoring invalid inputs" (for any wire-level inputs, the HSM will have wire-level outputs). Furthermore, similar to the case of program equivalence, some Knox specifications such as the password hasher have no instances that are trace-equivalent but not equal, so trace-equivalence preservation at the circuit level would be vacuous.

10 Conclusion

Information-preserving refinement (IPR) is a new security definition that captures the idea that a circuit-level implementation should implement its logical-level specification and leak nothing more. Knox demonstrates that IPR is useful in practice for ruling out bugs in an HSM's hardware and software. We believe that IPR is applicable beyond HSMs and hope that it can serve as a foundation of future security definitions.

Acknowledgments

We would like to thank Henry Corrigan-Gibbs and Joseph Tassarotti for insightful discussions that helped us develop the IPR definition. We are grateful to Emina Torlak and Xi Wang for their guidance on working with Rosette. This paper has been improved thanks to feedback from many individuals, including Akshay Narayan, Alexandra Henzinger, Ariel Szekely, Derek Leung, Kyle Hogan, Ralf Jung, Robert Morris, Sacha Servan-Schreiber, Stella Lau, Tej Chajed, Thomas Bourgeat, Yun-Sheng Chang, the anonymous reviewers, and our shepherd, George Candea. This research was supported by NSF award CNS-1812522 and by Google.

References

[1] CVE-2004-0320. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0320, Sept. 2004.

[2] YSA-2015-1. https://developers.yubico.com/ykneo-openpgp/SecurityAdvisory%202015-04-14.html, Apr. 2015.

[3] CVE-2018-6875. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6875, Feb. 2018.

[4] YSA-2018-01. https://www.yubico.com/support/security-advisories/ysa-2018-01/, Jan. 2018.

[5] CVE-2019-18671. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18671, Nov. 2019.

[6] CVE-2019-18672. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18672, Nov. 2019.

[7] YSA-2020-04. https://www.yubico.com/support/security-advisories/ysa-2020-04/, July 2020.

[8] CVE-2021-31616. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31616, Apr. 2021.

[9] WhatsApp security whitepaper: Security of end-to-end encrypted backups. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf, Sept. 2021.

[10] J. Aas, R. Barnes, B. Case, Z. Durumeric, P. Eckersley, A. Flores-López, J. A. Halderman, J. Hoffman-Andrews, J. Kasten, E. Rescorla, S. Schoen, and B. Warren. Let's Encrypt: An automated certificate authority to encrypt the entire web. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pages 2473–2487, London, United Kingdom, Nov. 2019.

[11] M. Abadi. *Protection in Programming-Language Translations*, pages 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48749-4.

[12] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM side-channel(s). In *Proceedings of the 2002 IACR Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Redwood City, CA, Aug. 2002.

[13] E. Alkassar, W. J. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In *Proceedings of the 3rd Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 71–85, Edinburgh, United Kingdom, Aug. 2010.

[14] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 1807–1823, Dallas, TX, Oct.–Nov. 2017.

[15] Apple, Inc. Apple platform security. https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf, May 2021.

[16] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 97–113, Huntsville, Ontario, Canada, Oct. 2019.

[17] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. SoK: Computer-aided cryptography. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, pages 777–795, Virtual conference, May 2021.

[18] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, pages 90–101, Savannah, GA, Jan. 2009.

[19] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st Annual International Cryptology Conference (CRYPTO)*, pages 71–90, Santa Barbara, CA, Aug. 2011.

[20] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving C compiler. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, LA, Jan. 2020.

[21] J.-B. Bédrune and G. Campana. Everybody be cool, this is a robbery! https://donjon.ledger.com/BlackHat2019-presentation/, Aug. 2019.

[22] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Security Symposium*, pages 207–201, Washington, DC, Aug. 2015.

[23] W. R. Bevier, W. A. Hunt Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, Dec. 1989.

[24] B. Blanchet. A computationally sound mechanized prover for security protocols. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 140–154, Oakland, CA, May 2006.

[25] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the 26th USENIX Security Symposium*, pages 917–934, Vancouver, Canada, Aug. 2017.

[26] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 260–269, Chicago, IL, Oct. 2010.

[27] B. Boston, S. Breese, J. Dodds, M. Dodds, B. Huffman, A. Petcher, and A. Stefanescu. Verified cryptographic code for everybody. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV)*, pages 645–668, Los Angeles, CA, July 2021.

[28] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, Las Vegas, NV, Oct. 2001.

[29] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang. Verifying Curve25519 software. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 299–309, Scottsdale, AZ, Nov. 2014.

[30] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 648–664, Santa Barbara, CA, June 2016.

[31] F. Cremonese. Security analysis of the Solo firmware. https://blog.doyensec.com/2020/02/19/solokeys-audit.html, Feb. 2020.

[32] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, Budapest, Hungary, Mar.–Apr. 2008.

[33] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, pages 331–344, Pittsburgh, PA, June 2008.

[34] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 73–90, San Francisco, CA, May 2019.

[35] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2021.

[36] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 287–305, Shanghai, China, Oct. 2017.

[37] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. https://racket-lang.org/tr1/.

[38] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy. A verified, efficient embedding of a verifiable assembly language. In *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*, Cascais, Portugal, Jan. 2019.

[39] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, Apr. 1982.

[40] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, New York, NY, May 1987.

[41] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC)*, pages 291–304, Providence, RI, May 1985.

[42] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, Oct. 2014.

[43] M. Hemmel, J. Meltzer, T. Pornin, K. Ryan, J. Samuel, D. Wong, R. Wood, and G. Worona. Android cloud backup/restore. https://research.nccgroup.com/wp-content/uploads/2020/07/Final_Public_Report_NCC_Group_Google_EncryptedBackup_2018-10-10_v1.0.pdf, Oct. 2018.

[44] M. Hutter and J.-M. Schmidt. The temperature side channel and heating fault attacks. In *Proceedings of the 12th Smart Card Research and Advanced Application Conference (CARDIS)*, pages 219–235, Berlin, Germany, Nov. 2013.

[45] J. Jancar, V. Sedlacek, P. Svenda, and M. Sys. Minerva: The curse of ECDSA nonces (systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces). *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):281–308, 2020.

[46] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, Network Working Group, Feb. 1997.

[47] I. Krstić. Behind the scenes with iOS security. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf, Aug. 2016.

[48] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1041–1053, Phoenix, AZ, June 2019.

[49] R. Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *Proceedings of the 2000 IACR Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 78–92, Worcester, MA, Aug. 2000.

[50] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the AFIPS 1968 Fall Joint Computer Conference*, pages 267–277, San Francisco, CA, Dec. 1968.

[51] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *Proceedings of the 29th USENIX Security Symposium*, pages 2057–2073, Aug. 2020.

[52] N. Moroze, A. Athalye, M. F. Kaashoek, and N. Zeldovich. rtlv: push-button verification of software on hardware. In *Proceedings of the 5th Workshop on Computer Architecture Research with RISC-V (CARRV)*, Virtual conference, June 2021.

[53] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. HOTP: An HMAC-based one-time password algorithm. RFC 4226, Network Working Group, Dec. 2005.

[54] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-based one-time password algorithm. RFC 6238, Network Working Group, May 2011.

[55] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013.

[56] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–147, Saint-Malo, France, Oct. 1997.

[57] National Institute of Standards and Technology. Secure hash standard. Federal Information Processing Standards (FIPS) 180-4, U.S. Department of Commerce, Washington, DC, Aug. 2015.

[58] OASIS PKCS 11 Technical Committee. PKCS #11 cryptographic token interface current mechanisms specification version 3.0. https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/os/pkcs11-curr-v3.0-os.html, June 2020.

[59] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF)*, pages 392–404, Santa Barbara, CA, Sept. 2017.

[60] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 51(6), Nov. 2019.

[61] A. Petcher and G. Morrisett. The Foundational Cryptography Framework. In *Proceedings of the 4th International Conference on Principles of Security and Trust*, pages 53–72, Apr. 2015.

[62] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Oxford, United Kingdom, Sept. 2017.

[63] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 983–1002, San Francisco, CA, May 2020.

[64] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 287–306, Carlsbad, CA, Oct. 2018.

[65] S. Srinivas, D. Balfanz, E. Tiffany, and A. Czeskis. Universal 2nd Factor (U2F) overview. https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-overview-v1.1-id-20160915.pdf, Sept. 2016.

[66] J. Strömbergson. sha256. https://github.com/secworks/sha256, 2013.

[67] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541, Edinburgh, United Kingdom, June 2014.

[68] F. Uekermann. Buggy OTP slot range check. https://github.com/Nitrokey/nitrokey-pro-firmware/issues/4, June 2016.

[69] C. X. Wolf. Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys, 2012.

[70] C. X. Wolf. Project IceStorm — Lattice iCE40 FPGAs bitstream documentaion. https://github.com/YosysHQ/icestorm, 2015.

[71] C. X. Wolf. PicoRV32 – a size-optimized RISC-V CPU. https://github.com/YosysHQ/picorv32, 2015.

[72] C. X. Wolf, gatecat, D. Gisselquist, S. Bazanski, M. Milanovic, and E. Hung. nextpnr. https://github.com/YosysHQ/nextpnr, 2018.

[73] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. Cryptology ePrint Archive, Report 2005/388, Oct. 2005.

[74] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

ORCA: A Distributed Serving System for Transformer-Based Generative Models

Gyeong-In Yu
Seoul National University

Joo Seong Jeong
Seoul National University

Geon-Woo Kim
FriendliAI
Seoul National University

Soojeong Kim
FriendliAI

Byung-Gon Chun*
FriendliAI
Seoul National University

Abstract

Large-scale Transformer-based models trained for generation tasks (e.g., GPT-3) have recently attracted huge interest, emphasizing the need for system support for serving models in this family. Since these models generate a next token in an autoregressive manner, one has to run the model multiple times to process an inference request where each iteration of the model generates a single output token for the request. However, existing systems for inference serving do not perform well on this type of workload that has a multi-iteration characteristic, due to their inflexible scheduling mechanism that cannot change the current batch of requests being processed; requests that have finished earlier than other requests in a batch cannot return to the client, while newly arrived requests have to wait until the current batch completely finishes.

In this paper, we propose iteration-level scheduling, a new scheduling mechanism that schedules execution at the granularity of iteration (instead of request) where the scheduler invokes the execution engine to run only a single iteration of the model on the batch. In addition, to apply batching and iteration-level scheduling to a Transformer model at the same time, we suggest selective batching, which applies batching only to a selected set of operations. Based on these two techniques, we have implemented a distributed serving system called ORCA, with additional designs for scalability to models with hundreds of billions of parameters. Our evaluation on a GPT-3 175B model shows that ORCA can significantly outperform NVIDIA FasterTransformer in terms of both latency and throughput: 36.9× throughput improvement at the same level of latency.

1 Introduction

Language generation tasks are becoming increasingly paramount to many types of applications, such as chatbot [9, 52], summarization [41,45,54], code generation [13], and caption generation [65,66]. Moreover, recent works published by AI21 Labs [37], DeepMind [26,48], Google [15,21,63], Meta Platforms [10,67], Microsoft [50], Microsoft & NVIDIA [59], and OpenAI [12] have reported that every language processing task, including translation [11,17], classification [20,53], question-answering [32,33,40] and more, can be cast as a language generation problem and have shown great improvements along this direction. The rise of generative models is not limited to the language domain; the AI community has also given growing interest to generation problems in other domains such as image, video, speech, or a mixture of multiple domains [19,38,51,62]. At the heart of generative models lies the Transformer architecture [60] and its variants [15,47–49]. By relying on the attention mechanism [60], Transformer models can learn better representations where each element of the sequence may have a direct connection with every other element, which was not possible in recurrent models [25].

To use generative models in real-world applications, we often delegate the inference procedure to a separate service responsible for ML inference serving. The growing demands for this service, which should provide inference results for client requests at low latency and high throughput, have facilitated the development of inference serving systems such as Triton Inference Server [7] and TensorFlow Serving [42]. These systems can use a separately-developed DNN *execution engine* to perform the actual tensor operations. For example, we can deploy a service for language generation tasks by using a combination of Triton and FasterTransformer [4], an execution engine optimized for the inference of Transformer-based models. In this case, Triton is mainly responsible for grouping multiple client requests into a batch, while FasterTransformer receives the batch from Triton and conducts the inference procedure in the batched manner.

Unfortunately, we notice that the existing inference systems, including both the serving system layer and the execution engine layer, have limitations in handling requests for Transformer-based generative models. Since these models are trained to generate a next token in an autoregressive manner, one should run the model as many times as the number of tokens to generate, while for other models like ResNet [24] and

*Corresponding author.

BERT [18] a request can be processed by running the model once. That is, in order to process a request to the generative model, we have to run multiple *iterations* of the model; each iteration generates a single output token, which is used as an input in the following iteration. Such multi-iteration characteristic calls into question the current design of inference systems, where the serving system schedules the execution of the engine at the granularity of request. Under this design, when the serving system dispatches a batch of requests to the engine, the engine returns inference results for the entire batch at once after processing all requests within the batch. As different client requests may require different numbers of iterations for processing, requests that have finished earlier than others in the batch cannot return to the client, resulting in an increased latency. Requests arrived after dispatching the batch also should wait for processing the batch, which can significantly increase the requests' queueing time.

In this paper, we propose to schedule the execution of the engine *at the granularity of iteration* instead of request. In particular, the serving system invokes the engine to run only a single iteration of the model on the batch. As a result, a newly arrived request can be considered for processing after waiting for only a single iteration of the model. The serving system checks whether a request has finished processing after every return from the engine – hence the finished requests can also be returned to the clients immediately.

Nevertheless, a noticeable challenge arises when we attempt to apply batching and the iteration-level scheduling at the same time. Unlike the canonical request-level scheduling, the proposed scheduling can issue a batch of requests where each request has so far processed a different number of tokens. In such a case, the requests to the Transformer model cannot be processed in the batched manner because the attention mechanism calls for non-batchable tensor operations whose input tensors have variable shapes depending on the number of processed tokens.

To address this challenge, we suggest to apply batching only to a selected set of operations, which we call *selective batching*. By taking different characteristics of operations into account, selective batching splits the batch and processes each request individually for the Attention[1] operation while applying batching to other operations of the Transformer model. We observe that the decision not to batch the executions of the Attention operation has only a small impact on efficiency. Since the Attention operation is not associated with any model parameters, applying batching to Attention has no benefit of reducing the amount of GPU memory reads by reusing the loaded parameters across multiple requests.

Based on these techniques, we design and implement ORCA, a distributed serving system for Transformer-based generative models. In order to handle large-scale models, ORCA adopts parallelization strategies including intra-layer and inter-layer model parallelism, which were originally developed by training systems [55, 58] for Transformer models. We also devise a new scheduling algorithm for the proposed iteration-level scheduling, with additional considerations for memory management and pipelined execution across workers.

We evaluate ORCA using OpenAI GPT-3 [12] models with various configurations, scaling up to 341B of parameters. The results show that ORCA significantly outperforms FasterTransformer [4], showing 36.9× throughput improvement at the same level of latency. While we use a language model as a driving example throughout the paper and conduct experiments only on language models, generative models in other domains can benefit from our approach as long as the models are based on the Transformer architecture and use the autoregressive generation procedure [19, 38, 51, 62].

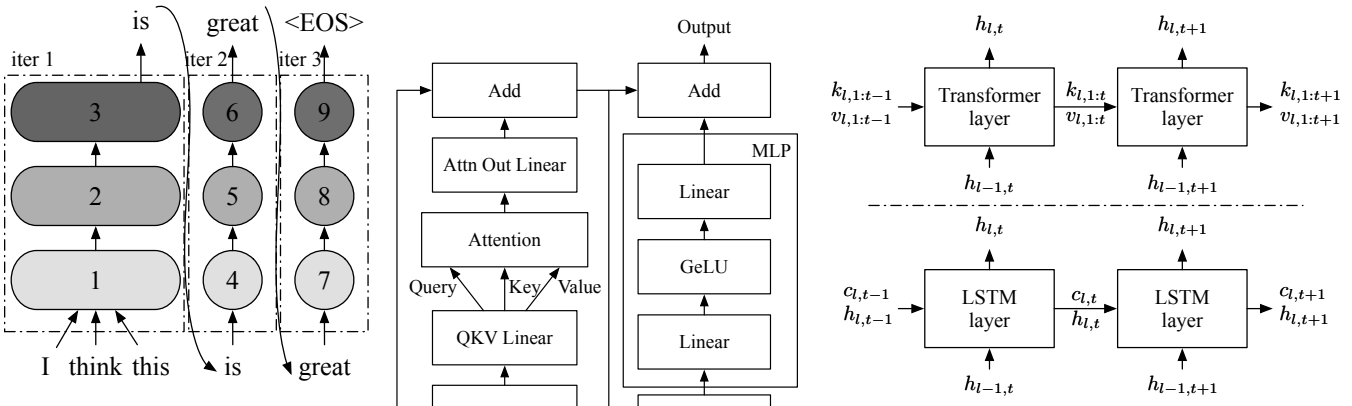
2 Background

We provide background on the inference procedure of GPT [12, 47], a representative example of Transformer-based generative models that we use throughout this paper, and ML inference serving systems.

Inference procedure of GPT. GPT is an autoregressive language model based on one of architectural variants of Transformer [60]. It takes text as input and produces new text as output. In particular, the model receives a sequence of input tokens and then completes the sequence by generating subsequent output tokens. Figure 1a illustrates a simplified computation graph that represents this procedure with a three-layer GPT model, where nodes and edges indicate Transformer layers and dependencies between the layers, respectively. The Transformer layers are executed in the order denoted by the numbers on the nodes, and the nodes that use the same set of model parameters (i.e., nodes representing the same layer) are filled with the same color.

The generated output token is fed back into the model to generate the next output token, imposing a sequential, one-by-one inference procedure. This autoregressive procedure of generating a single token is done by running all the layers of the model with the input, which is either a sequence of input tokens that came from the client or a previously generated output token. We define the run of all layers as an *iteration* of the model. In the example shown in Figure 1a, the inference procedure comprises three iterations. The first iteration ("iter 1") takes all the input tokens ("I think this") at once and generates the next token ("is"). This iteration composes an *initiation phase*, a procedure responsible for processing the input tokens and generating the first output token. The next two iterations ("iter 2" and "iter 3"), which compose an *increment phase*, take the output token of the preceding iteration and generate

[1] In some literature the Attention operation has an extended definition that includes linear layers (QKV Linear and Attn Out Linear; Figure 1b). On the other hand, we use a narrow definition as described in Figure 1b.



(a) A computation graph representing an inference procedure using a GPT model. The graph does not depict layers other than Transformer layers (e.g., embedding) for simplicity.

(b) A Transformer layer used in GPT.

(c) Internal state usage of Transformer. h, k, v, and c refer to layer input/output, Attention key, Attention value, and LSTM internal memory, respectively. l denotes layer index and t denotes token index.

Figure 1: Illustrations for GPT's inference procedure, Transformer layer, and internal state usage.

the next token. In this case, "iter 3" is the last iteration because it produces "<EOS>", a special end-of-sequence token that terminates output generation. Note that while the increment phase comprises multiple iterations because each iteration is only able to process a single token, the initiation phase is typically implemented as a single iteration by processing all the input tokens in parallel.

The original Transformer [60] employs two stacks of Transformer layers, while GPT's architecture consists of a single layer stack, namely decoder. Figure 1b shows a Transformer layer used in GPT. Among the operations that compose the Transformer layer, *Attention* is the essence that distinguishes Transformer from other architectures. At a high level, the Attention operation computes a weighted average of the tokens of interest so that each token in the sequence is aware of the other. It takes three inputs, query, key, and value, computes dot products of the query (for the current token) with all keys (for the tokens of interest), applies Softmax on the dot products to get weights, and conducts weighted average of all values associated with the weights.

Since the Attention requires keys and values of all preceding tokens,[2] we consider the keys and values as internal states that should be maintained across multiple iterations. A naïve, state-less inference procedure would take all tokens in the sequence (including both the client-provided input tokens and the output tokens generated so far) to recompute all the keys and values at every iteration. To avoid such recomputation, fairseq [43] suggests incremental decoding, which saves the keys and values for reuse in successive iterations. Other systems for Transformer such as FasterTransformer [4] and Megatron-LM [3] also do the same.

[2]Language models like GPT use causal masking, which means all preceding tokens are of interest and participate in the Attention operation.

Figure 1c illustrates the state usage pattern of Transformer, along with LSTM [25] that also maintains internal states. The main difference is that the size of the states (k for Attention key and v for value) in Transformer increases with iteration, whereas the size of the states (c for LSTM internal memory and h for LSTM layer's input/output) in LSTM remains constant. When processing the token at index t, the Attention operation takes all previous Attention keys $k_{l,1:t-1}$ and values $v_{l,1:t-1}$ along with the current key $k_{l,t}$ and value $v_{l,t}$.[3] Therefore, the Attention operation should perform computation on tensors of different shapes depending on the number of tokens already processed.

Prior to the Attention operation, there are the layer normalization operation (LayerNorm) and the QKV Linear (linear and split operations to get the query, key and value). Operations performed after Attention are, in order, a linear operation (Attn Out Linear), an add operation for residual connection (Add), layer normalization operation (LayerNorm), the multi-layer perceptron (MLP) operations, and the other residual connection operation (Add).

ML inference serving systems. Growing demands for ML-driven applications have made ML inference serving service a critical workload in modern datacenters. Users (either the end-user or internal microservices of the application) submit requests to an inference service, and the service gives replies on the requests based on a pre-defined ML model using its provisioned resource, typically equipped with specialized accelerators such as GPUs and TPUs. In particular, the service runs a DNN model with input data to generate output for the

[3]$k_{l,1:t-1}$ represents Attention keys of the l-th layer for tokens at indices 1 to $t-1$ while $k_{l,t}$ is for the Attention key of the l-th layer for the token at index t. Same for $v_{l,1:t-1}$ and $v_{l,t}$.

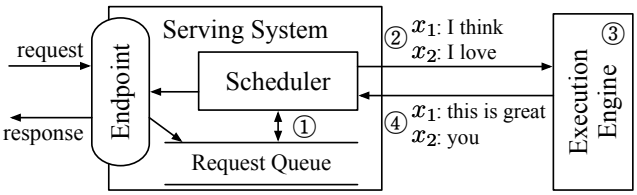


Figure 2: Overall workflow of serving a generative language model with existing systems.

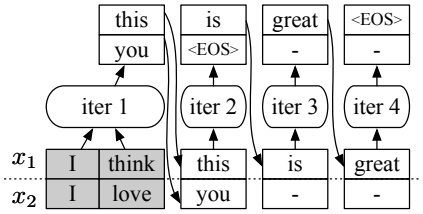


Figure 3: An illustration for a case where the requests have the same input length but some requests finish earlier than others. Shaded tokens represent input tokens. "-" denotes inputs and outputs of extra computation imposed by the scheduling.

request. Just like other services operating on datacenters, a well-managed inference service should provide low latency and high throughput within a reasonable amount of cost.

To meet such constraints, service operators often use ML inference serving systems such as Triton Inference Server [7] and TensorFlow Serving [42]. These systems can be seen as an abstraction sitting atop underlying model *execution engines* such as TensorRT [6], TVM [14], TensorFlow [8], and many others [44, 46], being agnostic to various kinds of ML models, execution engines, and computing hardware. While delegating the role of driving the main mathematical operations to the engines, serving systems are in charge of exposing endpoints that receive inference requests, scheduling executions of the engine, and sending responses to the requests. Accordingly, these systems focus on aspects such as batching the executions [7, 16, 35, 42, 56], selecting an appropriate model from multiple model variants [16, 27, 30, 57], deploying multiple models (each for different inference services) on the same device [7, 29, 35, 56], and so on.

Among the features and optimizations provided by serving systems, batching is a key to achieve high accelerator utilization when using accelerators like GPUs. When we run the execution engine with batching enabled, the input tensors from multiple requests coalesce into a single, large input tensor before being fed to the first operation of the model. Since the accelerators prefer large input tensors over small ones to better exploit the vast amount of parallel computation units, the engine's throughput is highly dependent on the batch size, i.e., the number of inference requests the engine processes together. Reusing the model parameters loaded from off-chip memory is another merit in batched execution, especially when the model involves memory-intensive operations.

Figure 2 shows an overall workflow of serving a generative language model with existing serving systems and execution engines. The main component of the serving system (e.g., Triton [7]) is the scheduler, which is responsible for ① creating a batch of requests by retrieving requests from a queue and ② scheduling the execution engine (e.g., FasterTransformer [4]) to process the batch. The execution engine ③ processes the received batch by running multiple iterations of the model being served and ④ returns the generated text back to the serving system. In Figure 2, the serving system schedules the engine to process two requests (x_1: "I think", x_2: "I love") in a batch and the engine generates "this is great" and "you" for requests x_1 and x_2, respectively.

3 Challenges and Proposed Solutions

In this section, we describe challenges in serving Transformer-based generative models and propose two techniques: iteration-level scheduling and selective batching.

C1: Early-finished and late-joining requests. One major limitation of existing systems is that the serving system and the execution engine interact with each other only when (1) the serving system schedules the next batch on an idle engine; or (2) the engine finishes processing the current batch. In other words, these systems are designed to schedule executions at *request* granularity; the engine maintains a batch of requests fixed until all requests in the batch finish. This can be problematic in the serving of generative models, since each request in a batch may require different number of iterations, resulting in certain requests finishing earlier than the others. In the example shown in Figure 3, although request x_2 finishes earlier than request x_1, the engine performs computation for both "active" and "inactive" requests throughout all iterations. Such extra computation for inactive requests (x_2 at iter 3 and 4) limits the efficiency of batched execution.

What makes it even worse is that this behavior prevents an early return of the finished request to the client, imposing a substantial amount of extra latency. This is because the engine only returns the execution results to the serving system when it finishes processing all requests in the batch. Similarly, when a new request arrives in the middle of the current batch's execution, the aforementioned scheduling mechanism makes the newly arrived request wait until all requests in the current batch have finished. We argue that the current request-level scheduling mechanism cannot efficiently handle workloads with multi-iteration characteristic. Note that this problem of early-finished and late-joining requests does not occur in the training of language models; the training procedure finishes processing the whole batch in a single iteration by using the teacher forcing technique [64].

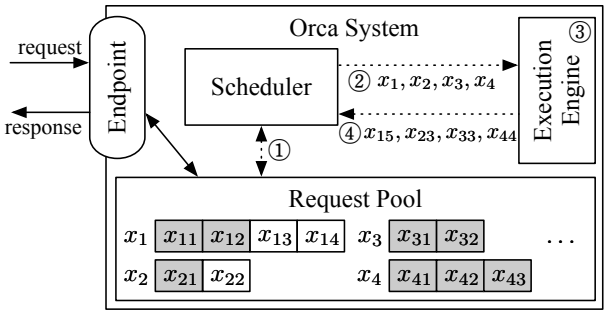


Figure 4: System overview of ORCA. Interactions between components represented as dotted lines indicate that the interaction takes place at every iteration of the execution engine. x_{ij} is the j-th token of the i-th request. Shaded tokens represent input tokens received from the clients, while unshaded tokens are generated by ORCA. For example, request x_1 initially arrived with two input tokens (x_{11}, x_{12}) and have run two iterations so far, where the first and second iterations generated x_{13} and x_{14}, respectively. On the other hand, request x_3 only contains input tokens (x_{31}, x_{32}) because it has not run any iterations yet.

S1: Iteration-level scheduling. To address the above limitations, we propose to schedule executions at the granularity of *iteration*. At high level, the scheduler repeats the following procedure: (1) selects requests to run next; (2) invokes the engine to execute *one iteration* for the selected requests; and (3) receives execution results for the scheduled iteration. Since the scheduler receives a return on every iteration, it can detect the completion of a request and immediately return its generated tokens to the client. For a newly arrived request, the request gets a chance to start processing (i.e., the scheduler may select the new request to run next) after execution of the currently scheduled iteration, significantly reducing the queueing delay. With iteration-level scheduling, the scheduler has a full control on how many and which requests are processed in each iteration.

Figure 4 depicts the system architecture and the overall workflow of ORCA using the iteration-level scheduling. ORCA exposes an *endpoint* (e.g., HTTPS or gRPC) where inference requests arrive at the system and responses to the requests are sent out. The endpoint puts newly arrived requests in the *request pool*, a component that manages all requests in the system during their lifetime. The pool is monitored by the *scheduler*, which is responsible for: selecting a set of requests from the pool, scheduling the *execution engine* to run an iteration of the model on the set, receiving execution results (i.e., output tokens) from the engine, and updating the pool by appending each output token to the corresponding request. The engine is an abstraction for executing the actual tensor operations, which can be parallelized across multiple GPUs spread across multiple machines. In the example shown in Figure 4, the scheduler ① interacts with the request pool to decide which requests to run next and ② invokes the engine to run four selected requests: (x_1, x_2, x_3, x_4). The scheduler provides the engine with input tokens of the requests scheduled for the first time. In this case, x_3 and x_4 have not run any iterations yet, so the scheduler hands over (x_{31}, x_{32}) for x_3 and (x_{41}, x_{42}, x_{43}) for x_4. The engine ③ runs an iteration of the model on the four requests and ④ returns generated output tokens $(x_{15}, x_{23}, x_{33}, x_{44})$, one for each scheduled request. Once a request has finished processing, the request pool removes the finished request and notifies the endpoint to send a response. Unlike the method shown in Figure 2 that should run multiple iterations on a scheduled batch until finish of all requests within the batch, ORCA's scheduler can change which requests are going to be processed at every iteration. We describe the detailed algorithm about how to select the requests at every iteration in Section 4.2.

C2: Batching an arbitrary set of requests. When we try to use the iteration-level scheduling in practice, one major challenge that we are going to face is batching. To achieve high efficiency, the execution engine should be able to process any selected set of requests in the batched manner. Without batching, one would have to process each selected request one by one, losing out on the massively parallel computation capabilities of GPUs.

Unfortunately, there is no guarantee that even for a pair of requests (x_i, x_j), for the next iteration, their executions can be merged and replaced with a batched version. There are three cases for a pair of requests where the next iteration cannot be batched together: (1) both requests are in the initiation phase and each has different number of input tokens (e.g., x_3 and x_4 in Figure 4); (2) both are in the increment phase and each is processing a token at different index from each other (x_1 and x_2); or (3) each request is in the different phase: initiation or increment (x_1 and x_3). Recall that in order to batch the execution of multiple requests, the execution of each request must consist of identical operations, each consuming identically-shaped input tensors. In the first case, the two requests cannot be processed in a batch because the "length" dimension of their input tensors, which is the number of input tokens, are not equal. The requests in the second case have difference in the tensor shape of Attention keys and values because each processes token at different index, as shown in Figure 1c. For the third case, we cannot batch the iterations of different phases because they take different number of tokens as input; an iteration of the initiation phase processes all input tokens in parallel for efficiency, while in the increment phase each iteration takes a single token as its input (we assume the use of fairseq-style incremental decoding [43]).

Batching is only applicable when the two selected requests are in the same phase, with the same number of input tokens (in case of the initiation phase) or with the same token index (in case of the increment phase). This restriction significantly reduces the likelihood of batching in real-world workloads,

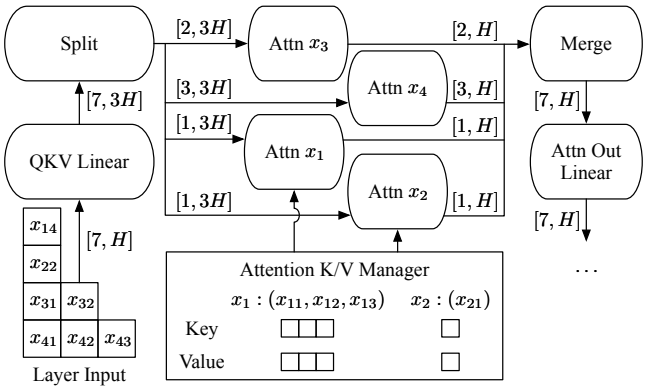


Figure 5: An illustration of ORCA execution engine running a Transformer layer on a batch of requests with selective batching. We only depict the QKV Linear, Attention, and Attention Out Linear operations for simplicity.

because the scheduler should *make a wish* for the presence of two requests eligible for batching at the same time. The likelihood further decreases exponentially as the batch size increases, making it impractical to use a large batch size that can pull out better throughput without compromising latency.

S2: Selective batching. We propose *selective batching*, a technique for batched execution that allows high flexibility in composing requests as a batch. Instead of processing a batch of requests by "batchifying" all tensor operations composing the model, this technique selectively apply batching only to a handful of operations.

The main problem regarding batching described above is that the three aforementioned cases[4] correspond to irregularly shaped input (or state) tensors, which cannot be coalesced into a single large tensor and fed into a batch operation. In the canonical batching mechanism, at each iteration, a Transformer layer takes a 3-dimensional input tensor of shape $[B,L,H]$ generated by concatenating multiple $[L,H]$ input tensors of requests in a batch, where B is the batch size, L is the number of tokens processed together, and H is the hidden size of the model. For example, in Figure 3, "iter 1" (initiation phase) takes an input tensor of shape $[2,2,H]$ and "iter 2" (increment phase) takes a tensor of shape $[2,1,H]$. However, when the scheduler decides to run an iteration on batch (x_1,x_2,x_3,x_4) in Figure 4, the inputs for requests in the initiation phase ($x_3 : [2,H]$ and $x_4 : [3,H]$) cannot coalesce into a single tensor of shape $[B,L,H]$ because x_3 and x_4 have different number of input tokens, 2 and 3.

Interestingly, not all operations are incompatible with such irregularly shaped tensors. Operations such as non-Attention matrix multiplication and layer normalization can be made to work with irregularly shaped tensors by flattening the tensors. For instance, the aforementioned input tensors for x_3 and x_4 can compose a 2-dimensional tensor of shape $[\sum L,H] = [5,H]$ without an explicit batch dimension. This tensor can be fed into all non-Attention operations including Linear, LayerNorm, Add, and GeLU operations because they do not need to distinguish tensor elements of different requests. On the other hand, the Attention operation requires a notion of requests (i.e., requires the batch dimension) to compute attention only between the tokens of the same request, typically done by applying cuBLAS routines for batch matrix multiplication.

Selective batching is aware of the different characteristics of each operation; it splits the batch and processes each request individually for the Attention operation while applying token-wise (instead of request-wise) batching to other operations without the notion of requests. Figure 5 presents the selective batching mechanism processing a batch of requests (x_1,x_2,x_3,x_4) described in Figure 4. This batch has 7 input tokens to process, so we make the input tensor have a shape of $[7,H]$ and apply the non-Attention operations. Before the Attention operation, we insert a `Split` operation and run the Attention operation separately on the split tensor for each request. The outputs of Attention operations are merged back into a tensor of shape $[7,H]$ by a `Merge` operation, bringing back the batching functionality to the rest of operations.

To make the requests in the increment phase can use the Attention keys and values for the tokens processed in previous iterations, ORCA maintains the generated keys and values in the *Attention K/V manager*. The manager maintains these keys and values separately for each request until the scheduler explicitly asks to remove certain request's keys and values, i.e., when the request has finished processing. The Attention operation for request in the increment phase (x_1 and x_2) takes keys and values of previous tokens (x_{11},x_{12},x_{13} for x_1; x_{21} for x_2) from the manager, along with the current token's query, key, and value from the Split operation to compute attention between the current token and the previous ones.

4 ORCA Design

Based on the above techniques, we design and implement ORCA: a distributed serving system for Transformer-based generative models. We have already discussed the system components and the overall execution model of ORCA while describing Figure 4. In this section, we answer the remaining issues about how to build an efficient system that can scale to large-scale models with hundreds of billions of parameters. We also describe the scheduling algorithm for iteration-level scheduling, i.e., how to select a batch of requests from the request pool at every iteration.

4.1 Distributed Architecture

Recent works [12, 31] have shown that scaling language models can dramatically improve the quality of models. Hence,

[4] We use the first case as a driving example, but the argument can be similarly applied to the other two cases.

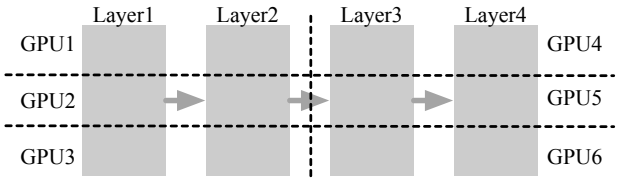


Figure 6: An example of intra- and inter- layer parallelism. A vertical dotted line indicates partitioning between layers and a horizontal line indicates partitioning within a layer.

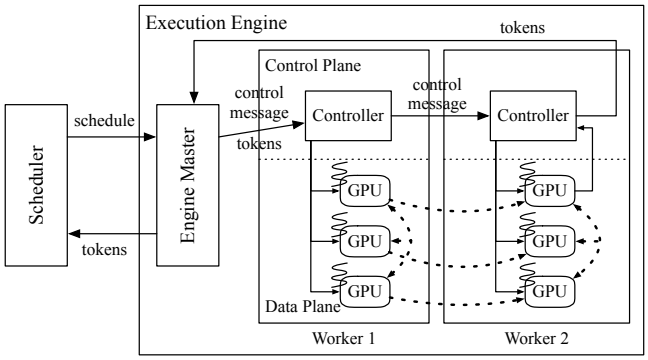


Figure 7: An illustration of the distributed architecture of ORCA's execution engine using the parallelization configuration shown in Figure 6. For example, the first inter-layer partition (Layer1 and Layer2) in Figure 6 is assigned to Worker1, while the second partition is assigned to Worker2.

system support for serving such large language models is getting more importance, especially when the model does not fit in a single GPU. In such a case, one should split the model parameters along with the corresponding computation and distribute them across multiple GPUs and machines.

ORCA composes known parallelization techniques for Transformer models: intra-layer parallelism and inter-layer parallelism. These two model parallelism strategies, which are also used by FasterTransformer [4], have been originally developed for distributed training. Intra-layer parallelism [55, 58] splits matrix multiplications (i.e., Linear and Attention operations) and their associated parameters over multiple GPUs. We omit the detail about how this strategy partitions each matrix multiplication. On the other hand, inter-layer parallelism splits Transformer layers over multiple GPUs. ORCA assigns the same number of Transformer layers to each GPU. Figure 6 illustrates an example application of intra- and inter- layer parallelism to a 4-layer GPT model. The 4 layers are split into 2 inter-layer partitions, and the layers in the partition are subdivided into 3 intra-layer partitions. We assign each partition to a GPU, using a total of 6 GPUs.

The ORCA execution engine supports distributed execution using the techniques described above. Figure 7 depicts the architecture of an ORCA engine. Each *worker process* is responsible for an inter-layer partition of the model and can be placed on a different machine from each other. In particular, each worker manages one or more CPU threads each dedicated for controlling a GPU, the number of which depends on the degree of intra-layer parallelism.

The execution procedure of the ORCA execution engine is as follows. Once the engine is scheduled to run an iteration of the model for a batch of requests, the *engine master* forwards the received information about the scheduled batch to the first *worker process* (Worker1). The information includes tokens for the current iteration and a control message, which is composed of ids of requests within the batch, current token index (for requests in the increment phase), and number of input tokens (for requests in the initiation phase). The *controller* of Worker1 hands over the information received from the engine master to the GPU-controlling threads, where each thread parses the information and issues proper GPU kernels to its associated GPU. For example, the kernel for the Attention operation uses the request id and the current token index to get the GPU memory address of previous keys and values kept by the Attention K/V manager. In the meantime, the controller also forwards the control message to the controller of the next worker (Worker2), without waiting for the completion of the kernels issued on the GPUs of Worker1. Unlike Worker1, the controller of the last worker (Worker2) waits for (i.e., synchronize with) the completion of the issued GPU kernels, in order to fetch the output token for each request and send the tokens back to the engine master.

To keep GPUs busy as much as possible, we design the ORCA engine to minimize synchronization between the CPU and GPUs. We observe that current systems for distributed inference (e.g., FasterTransformer [4] and Megatron-LM [3]) have CPU-GPU synchronization whenever each process receives control messages[5] because they exchange the messages through a GPU-to-GPU communication channel – NCCL [5]. The exchange of these control messages occurs at every iteration, imposing a non-negligible performance overhead. On the other hand, ORCA separates the communication channels for control messages (plus tokens) and tensor data transfer, avoiding the use of NCCL for data used by CPUs. Figure 7 shows that the ORCA engine uses NCCL exclusively for exchanging intermediate tensor data (represented by dashed arrows) as this data is produced and consumed by GPUs. Control messages, which is used by the CPU threads for issuing GPU kernels, sent between the engine master and worker controllers by a separate communication channel that does not involve GPU such as gRPC [2].

4.2 Scheduling Algorithm

The ORCA scheduler makes decisions on which requests should be selected and processed at every iteration. The scheduler has high flexibility in selecting a set of requests to com-

[5]This includes various metadata such as batch size, sequence length, and whether a request within the batch has finished processing.

pose a batch, because of the selective batching technique that allows the engine to run any set of requests in the batched manner. Now the main question left is how to select the requests at every iteration.

We design the ORCA scheduler to use a simple algorithm that does not change the processing order of client requests; early-arrived requests are processed earlier. That is, we ensure iteration-level first-come-first-served (FCFS) property. We define the iteration-level FCFS property for workloads with multi-iteration characteristics as follows: for any pair of requests (x_i, x_j) in the request pool, if x_i has arrived earlier than x_j, x_i should have run the same or more iterations than x_j. Note that some late-arrived requests may return earlier to clients if the late request requires a smaller number of iterations to finish.

Still, the scheduler needs to take into account additional factors: diminishing returns to increasing the batch size and GPU memory constraint. Increasing the batch size trades off increased throughput for increased latency, but as the batch size becomes larger, the amount of return (i.e., increase in throughput) diminishes. Therefore, just like other serving systems [7, 16], ORCA also has a notion of a max batch size: the largest possible number of requests within a batch. The ORCA system operator can tune this knob to maximize throughput while satisfying one's latency budget. We will discuss this in more details with experiment results in Section 6.2.

Another factor is the GPU memory constraint. Optimizing memory usage by reusing buffers for intermediate results across multiple operations is a well-known technique used by various systems [4, 6], and ORCA also adopts this technique. However, unlike the buffers for intermediate results that can be reused immediately, buffers used by the Attention K/V manager for storing the keys and values cannot be reclaimed until the ORCA scheduler notifies that the corresponding request has finished processing. A naïve implementation can make the scheduler fall into a deadlock when the scheduler cannot issue an iteration for any requests in the pool because there is no space left for storing a new Attention key and value for the next token. This requires the ORCA scheduler to be aware of the remaining size of pre-allocated memory regions for the manager.

The ORCA scheduler takes all these factors into account: it selects at most "max batch size" requests based on the arrival time, while reserving enough space for storing keys and values to a request when the request is scheduled for the first time. We describe the scheduling process in Algorithm 1. The algorithm selects a batch of requests from the request pool (line 4) and schedules the batch (line 5). The *Select* function (line 17) selects at most *max_bs* requests from the pool based on the arrival time of the request (lines 20-22). Algorithm 1 does not depict the procedure of request arrival and return; one may think of it as there exist concurrent threads inserting newly arrived requests into *request_pool* and removing finished requests from *request_pool*.

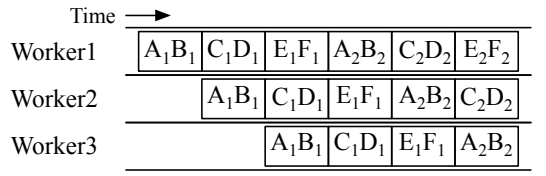
Algorithm 1: ORCA scheduling algorithm

```
   Params: n_workers: number of workers, max_bs:
           max batch size, n_slots: number of K/V slots
1  n_scheduled ← 0
2  n_rsrv ← 0
3  while true do
4      batch, n_rsrv ← Select(request_pool, n_rsrv)
5      schedule engine to run one iteration of
       the model for the batch
6      foreach req in batch do
7          req.state ← RUNNING
8      n_scheduled ← n_scheduled + 1
9      if n_scheduled = n_workers then
10         wait for return of a scheduled batch
11         foreach req in the returned batch do
12             req.state ← INCREMENT
13             if finished(req) then
14                 n_rsrv ← n_rsrv − req.max_tokens
15         n_scheduled ← n_scheduled − 1
16
17 def Select(pool, n_rsrv):
18     batch ← {}
19     pool ← {req ∈ pool | req.state ≠ RUNNING}
20     SortByArrivalTime(pool)
21     foreach req in pool do
22         if batch.size() = max_bs then break
23         if req.state = INITIATION then
24             new_n_rsrv ← n_rsrv + req.max_tokens
25             if new_n_rsrv > n_slots then break
26             n_rsrv ← new_n_rsrv
27         batch ← batch ⋃ {req}
28     return batch, n_rsrv
```

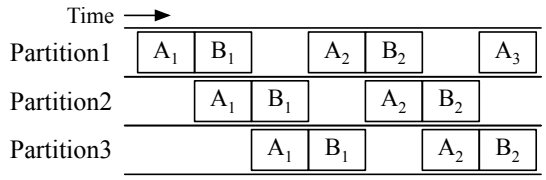
When the scheduler considers a request in the initiation phase, meaning that the request has never been scheduled yet, the scheduler uses the request's *max_tokens*[6] attribute to reserve *max_tokens* slots of GPU memory for storing the keys and values in advance (lines 23-26). The scheduler determines whether the reservation is possible (line 25) based on *n_rsrv*, the number of currently reserved slots, where a slot is defined by the amount of memory required for storing an Attention key and value for a single token. Here, *n_slots* is a parameter tuned by the ORCA system operator indicating the size of memory region (in terms of slots) allocated to the Attention K/V manager. Since the number of tokens in a request cannot exceed *max_tokens*, if the reservation is possible, it is guaranteed that the manager can allocate buffers for the newly generated keys and values until the request finishes.

Unlike the tuning of *max_bs* that requires quantifying the trade-off between latency and throughput, the ORCA system

[6] The *max_tokens* attribute is a per-request option, meaning the maximum number of tokens that a request can have after processing.



(a) ORCA execution pipeline.



(b) FasterTransformer execution pipeline.

Figure 8: Comparison of the use of pipeline parallelism in ORCA and FasterTransformer where X_i is the i-th iteration of request X.

operator can easily configure *n_slots* without any experiments. Given a model specification (e.g., hidden size, number of layers, etc.) and degrees of intra- and inter- layer parallelism, ORCA's GPU memory usage mostly depends on *n_slots*. That is, the operator can simply use the largest possible *n_slots* under the memory constraint.

Pipeline parallelism. ORCA's scheduler makes the execution of workers in the engine to be pipelined across multiple batches. The scheduler does not wait for the return of a scheduled batch until *n_scheduled*, the number of currently scheduled batches, reaches *n_workers* (line 9-10 of Algorithm 1). By doing so, the scheduler keeps the number of concurrently running batches in the engine to be *n_workers*, which means that every worker in the engine is processing one of the batches without being idle.

Figure 8a depicts the execution pipeline of 3 ORCA workers, using a max batch size of 2. We assume that the request A arrives before B, which arrives before C, and so on. At first, the scheduler selects requests A and B based on the arrival time and schedules the engine to process a batch of requests A and B (we call this batch AB), where Worker1, Worker2, and Worker3 process the batch in turn. The scheduler waits for the return of the batch AB only after the scheduler injects two more batches: CD and EF. Once the batch AB returns, requests A and B get selected and scheduled once again, because they are the earliest arrived requests among the requests in the pool.

In contrast, the interface between current serving systems and execution engines (e.g., a combination of Triton [7] and FasterTransformer [4]) does not allow injecting another batch before the finish of the current running batch, due to the request-level scheduling. That is, Triton cannot inject the next request C to FasterTransformer until the current batch AB finishes. To enable pipelined execution of multiple inter-layer partitions under such constraint, FasterTransformer splits a batch of requests into multiple *microbatches* [28] and pipelines the executions of partitions across the microbatches. In Figure 8b, FasterTransformer splits the batch AB into two microbatches, A and B. Since each partition processes a microbatch (which is smaller than the original batch) in the batched manner, the performance gain from batching can become smaller. Moreover, this method may insert *bubbles* into the pipeline when the microbatch size is too large, making the number of microbatches smaller than the number of partitions. While FasterTransformer needs to trade batching efficiency (larger microbatch size) for pipelining efficiency (fewer pipeline bubbles), ORCA is free of such a tradeoff – thanks to iteration-level scheduling – and can easily pipeline requests without dividing a batch into microbatches.

# Params	# Layers	Hidden size	# Inter-partitions	# Intra-partitions
13B	40	5120	1	1
101B	80	10240	1	8
175B	96	12288	2	8
341B	120	15360	4	8

Table 1: Configurations of models used in the experiments.

5 Implementation

We have implemented ORCA with 13K lines of C++, based on the CUDA ecosystem. We use gRPC [2] for the communication in the control plane of the ORCA engine, while NCCL [5] is used in the data plane, for both inter-layer and intra-layer communication. Since we design ORCA to focus on Transformer-based generative models, ORCA provides popular Transformer layers as a building block of models including the original encoder-decoder Transformer [60], GPT [47], and other variants discussed in Raffel et al. [49].

We have also implemented fused kernels for LayerNorm, Attention, and GeLU operators, just like other systems for training or inference of Transformer models [1, 4, 58]. For example, the procedure of computing dot products between Attention query and keys, Softmax on the dot products, and weighted average of Attention values are fused into a single CUDA kernel for the Attention operator. In addition, we go one step further and fuse the kernels of the split Attention operators by simply concatenating all thread blocks of the kernels for different requests. Although this fusion makes the thread blocks within a kernel have different characteristics and lifetimes (which is often discouraged by CUDA programming practice) because they process tensors of different shapes, we find this fusion to be beneficial by improving GPU utilization and reducing the kernel launch overhead [34, 39].

6 Evaluation

In this section, we present evaluation results to show the efficiency of ORCA.

Environment. We run our evaluation on Azure ND96asr A100 v4 VMs, each equipped with 8 NVIDIA 40-GB A100 GPUs connected over NVLink. We use at most four VMs depending on the size of the model being tested. Each VM has 8 Mellanox 200Gbps HDR Infiniband adapters, providing an 1.6Tb/s of interconnect bandwidth between VMs.

Models. Throughout the experiments, we use GPT [12] as a representative example of Transformer-based generative models. We use GPT models with various configurations, which is listed in Table 1. The configurations for 13B and 175B models come from the GPT-3 paper [12]. Based on these two models, we change the number of layers and hidden size to make configurations for 101B and 341B models. All models have a maximum sequence length of 2048, following the setting of the original literature [12]. We use fp16-formatted model parameters and intermediate activations for the experiments. We also apply inter- and intra- layer parallelism strategies described in Section 4.1, except for the 13B model that can fit in a GPU. For example, the 175B model is partitioned over a total of 16 GPUs by using 2 inter-layer partitions subdivided into 8 intra-layer partitions, where the 8 GPUs in the same VM belongs to the same inter-layer partition.

Baseline system. We compare with FasterTransformer [4], an inference engine that supports large scale Transformer models via distributed execution. While there exist other systems with the support for distributed execution such as Megatron-LM [3] and DeepSpeed [1], these systems are primarily designed and optimized for training workloads, which makes them show relatively lower performance compared to the inference-optimized systems.

Scenarios. We use two different scenarios to drive our evaluation. First, we design a microbenchmark to solely assess the performance of the ORCA engine without being affected by the iteration-level scheduling. In particular, we do not run the ORCA scheduler in this scenario. Instead, given a batch of requests, the testing script repeats injecting the same batch into the ORCA engine until all requests in the batch finishes, mimicking the behavior of the canonical request-level scheduling. We also assume that all requests in the batch have the same number of input tokens and generate the same number of output tokens. We report the time taken for processing the batch (not individual requests) and compare the result with FasterTransformer [4].

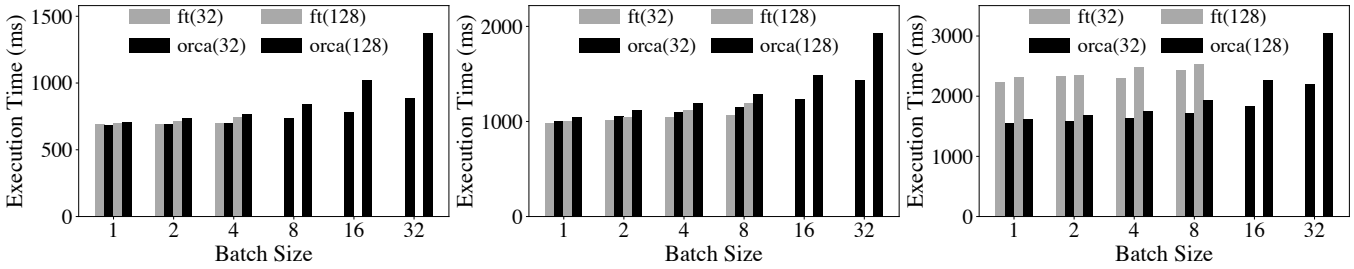
The second scenario tests the end-to-end performance of ORCA by emulating a workload. We synthesize a trace of client requests because there is no publicly-available request trace for generative language models. Each request in the synthesized trace is randomly generated by sampling the number of input tokens and a *max_gen_tokens* attribute, where the number of input tokens plus *max_gen_tokens* equals to the *max_tokens* attribute described in Section 4.2. We assume that all requests continue generation until the number of generated tokens reaches *max_gen_tokens*. In other words, we make the model never emit the "<EOS>" token. This is because we have neither the actual model checkpoint nor the actual input text so we do not have any information to guess the right timing of the "<EOS>" token generation. Once the requests are generated, we synthesize the trace by setting the request arrival time based on the Poisson process. To assess ORCA's behavior under varying load, we change the Poisson parameter (i.e., arrival rate) and adjust the request arrival time accordingly. We report latency and throughput using multiple traces generated from different distributions for better comparison and understanding of the behavior of ORCA and FasterTransformer.

6.1 Engine Microbenchmark

We first compare the performance of FasterTransformer and the ORCA engine using the first scenario. We set all requests in the batch to have the same number of input tokens (32 or 128) and generate 32 tokens. That is, in this set of experiments, all requests within the batch start and finish processing at the same time. We conduct experiments using three different models: 13B, 101B, and 175B. For each model, we use the corresponding parallelization strategy shown in Table 1.

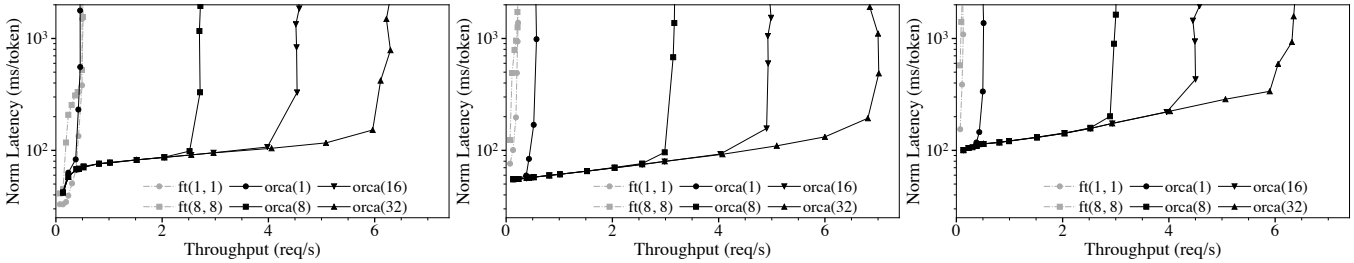
Figure 9 shows the performance of FasterTransformer and the ORCA engine for processing a batch composed of the same requests. In Figure 9a, the ORCA engine shows a similar (or slightly worse) performance compared to FasterTransformer across all configurations. This is because ORCA does not apply batching to the Attention operations, while FasterTransformer apply batching to all operations. Still, the performance difference is relatively small. Despite not batching the Attention operation, the absence of model parameters in Attention makes this decision has little impact on efficiency as there is no benefit of reusing model parameters across multiple requests.

Figure 9b presents similar results for the 101B model that uses all of the 8 GPUs in a single VM. From these results, we can say that the ORCA engine and FasterTransformer have comparable efficiencies in the implementations of CUDA kernels and the communication between intra-layer partitions. Note that FasterTransformer cannot use a batch size of 8 or larger with the 13B model (16 or larger with the 101B model) because of the fixed amount of memory pre-allocation for each request's Attention keys and values, which grows in proportion to the max sequence length of the model (2048 for this case). In contrast, ORCA avoids redundant memory



(a) 13B model, 1 GPU. (b) 101B model, 8 GPUs. (c) 175B model, 16 GPUs.

Figure 9: Execution time of a batch of requests using FasterTransformer and the ORCA engine without the scheduling component. Label "ft(*n*)" represents results from FasterTransformer processing requests with *n* input tokens. Configurations that incurs out of memory error are represented as missing entries (e.g., ft(32) for the 101B model with a batch size of 16).



(a) 101B model, 8 GPU. (b) 175B model, 16 GPUs. (c) 341B model, 32 GPUs.

Figure 10: Median end-to-end latency normalized by the number of generated tokens and throughput. Label "orca(*max_bs*)" represents results from ORCA with a max batch size of *max_bs*. Label "ft(*max_bs*, *mbs*)" represents results from FasterTransformer with a max batch size of *max_bs* and a microbatch size of *mbs*.

allocation by setting the size of buffers for the keys and values separately for each request based on the *max_tokens* attribute.

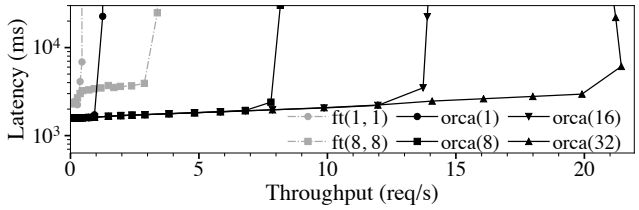
Next, we go one step further and experiment with the 175B model, which splits the layers into two inter-layer partitions. In this case, for better comparison, we disable pipelined execution of the inter-layer partitions for both systems. For FasterTransformer, we set the size of a microbatch to be equal to the batch size to disable pipelining. As shown in Figure 9c, the ORCA engine outperforms FasterTransformer by up to 47%. We attribute this performance improvement to the control-data plane separation described in Section 4.1. We omit the 341B model as it has similar results compared to the 175B model.

6.2 End-to-end Performance

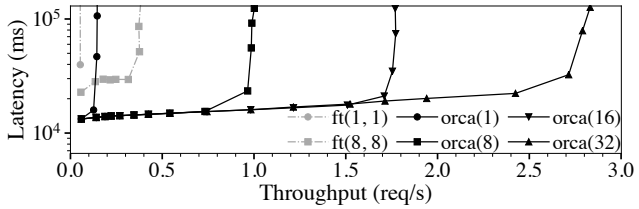
Now we assess the end-to-end performance of ORCA by measuring the latency and throughput with the synthesized request trace under varying load. When synthesizing the trace, we sample each request's number of input tokens from $U(32,512)$, a uniform distribution ranging from 32 to 512 (inclusive). The *max_gen_tokens* attributed is sampled from $U(1,128)$, which means that the least and the most time-consuming requests require 1 and 128 iterations of the model for processing, respectively.

Unlike the microbenchmark shown in Section 6.1, to measure the end-to-end performance, we test the entire ORCA software stack including the ORCA scheduler. Client requests arrive to the ORCA scheduler following the synthesized trace described above. We report results from various max batch size configurations. For FasterTransformer that does not have its own scheduler, we implement a custom scheduler that receives client requests, creates batches, and injects the batches to an instance of FasterTransformer. We make the custom scheduler create batches dynamically by taking at most max batch size requests from the request queue, which is the most common scheduling algorithm used by existing serving systems like Triton [7] and TensorFlow Serving [42]. Again, we report results from various max batch size configurations, along with varying microbatch sizes, an additional knob in FasterTransformer that governs the pipelining behavior (see Section 4.2).

Figure 10 shows median end-to-end latency and throughput. Since each request in the trace requires different processing time, which is (roughly) in proportion to the number of generated tokens, we report median latency normalized by the number of generated tokens of each request. From the figure, we can see that ORCA provides significantly higher throughput and lower latency than FasterTransformer. The only exception is the 101B model under low load (Figure 10a). In this



(a) (# in, # gen) = (32, 32)



(b) (# in, # gen) = (256, 256)

Figure 11: Median end-to-end latency and throughput, using the 175B model with traces composed of homogeneous requests. We do not normalize the latency since all requests have the same characteristic.

case, both ORCA and FasterTransformer do not have enough number of requests to process in a batch. That is, the latency will mostly depend on the engine's performance, which is shown in Figure 9b. As the load becomes heavier, ORCA provides higher throughput with a relatively small increase in latency, because the ORCA scheduler makes late-arrived requests hitch a ride with the current ongoing batch. In contrast, FasterTransformer fails to efficiently handle multiple requests that (1) arrive at different times; (2) require different number of iterations to finish; or (3) start with different number of input tokens, resulting in a peak throughput of 0.49 req/s and much higher latency. If we use the 175B or 341B model (Figures 10b and 10c) that employs more than one inter-layer partitions, ORCA outperforms FasterTransformer under every level of load in terms of both latency and throughput, resulting in an order of magnitude higher throughput when we compare results at a similar level of latency. For example, to match a median normalized latency of 190ms for the 175B model, which is a double of the normalized execution time (by the number of generated tokens) of "orca(128)" shown in Figure 9c, FasterTransformer provides a throughput of 0.185 req/s whereas ORCA provides a throughput of 6.81 req/s, which is a 36.9× speedup.

Varying batch size configurations. Figure 10 shows that the increase of the max batch size of ORCA results in a higher throughput without affecting the latency. This is because the iteration-level scheduling of ORCA resolves the problem of early-finished and late-joining requests. Nevertheless, there is no guarantee that increasing the batch size will not negatively affect the latency, for arbitrary hardware settings, models, and workloads. As mentioned in Section 4.2, the max batch size must be set carefully by considering both the required latency and throughput requirements.

Interestingly, larger max batch size in FasterTransformer does not necessarily help improving throughput. By testing all possible combinations of max batch size (*max_bs*) and microbatch size (*mbs*) on all models under varying load, we find that (*max_bs*, *mbs*) = (1, 1) or (8, 8) are the best options. Per our discussion in Section 4.1, FasterTransformer's microbatch-based pipelining can be less efficient because the engine is going to process at most *mbs* number of requests in the batched manner, which explains why the configurations with the maximum possible *mbs* (which is the same as *max_bs*) have better performance than others. In addition, while increasing *max_bs* can improve performance due to the increased batch size, at the same time, this also increases the likelihood of batching requests with large difference in the number of input tokens or the number of generated tokens. In such cases, FasterTransformer cannot efficiently handle the batch because (1) for the first iteration of the batch, FasterTransformer processes requests as if they all had the same input length as the shortest one; and (2) early-finished requests cannot immediately return to the clients.

Trace of homogeneous requests. We test the behavior of ORCA and FasterTransformer when using a trace of homogeneous requests, i.e., all requests in a trace have the same number of input tokens and the same *max_gen_tokens* attribute. Since all requests require the same number of iterations to finish processing, the problem of early-leaving requests does not occur for this trace. As a result, now the increase of the *max_bs* has a noticeable positive impact on the performance of FasterTransformer, as shown in Figure 11. Still, ORCA outperforms FasterTransformer (*max_bs*=8) except for the case using a max batch size of 1, where ORCA degenerates into a simple pipeline of the ORCA workers that does not perform batching.

7 Related Work and Discussion

Fine-grained batching for recurrent models. We would like to highlight BatchMaker [23] as one of the most relevant previous works. BatchMaker is a serving system for RNNs that performs scheduling and batching at the granularity of RNN cells, motivated by the unique RNN characteristic of repeating the same computation. Once a request arrives, BatchMaker breaks the dataflow graph for processing the request into RNN cells, schedules execution at the granularity of cells (instead of the entire graph), and batches the execution of identical cells (if any). Since each RNN cell always performs the exact same computation, BatchMaker can execute multiple RNN cells in a batched manner regardless of the position (i.e., token index) of the cell. By doing so, BatchMaker allows a newly arrived request for RNN to join (or a finished request

to leave) the current executing batch without waiting for the batch to completely finish.

However, BatchMaker cannot make batches of cells for Transformer models because there are too many distinct cells (a subgraph that encapsulates the computation for processing a token; Figure 1c) in the graph. Each cell at a different token index t must use a different set of Attention Keys/Values. As the cell for each t is different, the graph comprises L different cells (L denotes the number of input and generated tokens), significantly lowering the likelihood of cells of the same computation being present at a given moment (e.g., in Figure 10, L ranges from $33 = 32 + 1$ to $640 = 512 + 128$). Thus execution of the cells will be mostly serialized, making BatchMaker fall back to non-batched execution. BatchMaker also lacks support for large models that require model and pipeline parallelism.

While BatchMaker is geared towards detecting and aligning batch-able RNN cells, our key principle in designing ORCA is to perform as much computation as possible per each round of model parameter read. This is based on the insight that reading parameters from GPU global memory is a major bottleneck in terms of end-to-end execution time, for large-scale models. Adhering to this principle, we apply iteration-level scheduling and selective batching to process all "ready" tokens in a single round of parameter read, regardless of whether the processing of tokens can be batched (non-Attention ops) or not (Attention ops).

Specialized execution engines for Transformer models. The outstanding performance of Transformer-based models encourages the development of inference systems specialized for them. FasterTransformer [4], LightSeq [61], TurboTransformers [22] and EET [36] are such examples. Each of these systems behave as an backend execution engine of existing serving systems like Triton Inference Server [7] and TensorFlow Serving [42]. That is, these systems delegate the role of scheduling to the serving system layer, adhering to the canonical request-level scheduling. Instead, ORCA suggests to schedule executions at a finer granularity, which is not possible in current systems without changing the mechanism for coordination between the scheduler and the execution engine. Note that among these systems, FasterTransformer is the only one with the support for distributed execution. While systems like Megatron-LM [3] and DeepSpeed [1] can also be used for distributed execution, these systems are primarily optimized for large-scale training rather than inference serving.

Interface between serving systems and execution engines. Current general-purpose serving systems such as Triton Inference Server [7] and Clipper [16] serve as an abstraction for handling client requests and scheduling executions of the underlying execution engines. This approach is found to be beneficial by separating the design and implementation of the serving layer and the execution layer. However, we find that the prevalent interface between the two layers is too restricted for handling models like GPT [12], which has the multi-iteration characteristic. Instead, we design ORCA to tightly integrate the scheduler and the engine, simplifying the application of the two proposed techniques: iteration-level scheduling and selective batching. While in this paper we do not study a general interface design that supports the two techniques without losing the separation of abstractions, it can be an interesting topic to explore such possibility; we leave this issue to future work.

8 Conclusion

We present iteration-level scheduling with selective batching, a novel approach that achieves low latency and high throughput for serving Transformer-based generative models. Iteration-level scheduling makes the scheduler interact with the execution engine at the granularity of iteration instead of request, while selective batching enables batching arbitrary requests processing tokens at different positions, which is crucial for applying batching with iteration-level scheduling. Based on these techniques, we have designed and implemented a distributed serving system named ORCA. Experiments show the effectiveness of our approach: ORCA provides an order of magnitude higher throughput than current state-of-the-art systems at the same level of latency.

Acknowledgments

We thank our shepherd Amar Phanishayee and the anonymous reviewers for their insightful comments. This work was supported by FriendliAI Inc.

References

[1] DeepSpeed. Retrieved Dec 13, 2021 from https://github.com/microsoft/DeepSpeed.

[2] gRPC. Retrieved Dec 13, 2021 from https://grpc.io.

[3] Megatron-LM. Retrieved Dec 13, 2021 from https://github.com/NVIDIA/Megatron-LM.

[4] NVIDIA FasterTransformer. Retrieved Dec 13, 2021 from https://github.com/NVIDIA/FasterTransformer.

[5] NVIDIA NCCL. Retrieved Dec 13, 2021 from https://github.com/NVIDIA/nccl.

[6] NVIDIA TensorRT. Retrieved Dec 13, 2021 from https://developer.nvidia.com/tensorrt.

[7] NVIDIA Triton Inference Server. Retrieved Dec 13, 2021 from https://developer.nvidia.com/nvidia-triton-inference-server.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 265–283, 2016.

[9] Daniel Adiwardana, Minh-Thang Luong, David R So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, et al. Towards a Human-like Open-Domain Chatbot. *arXiv preprint arXiv:2001.09977*, 2020.

[10] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Ves Stoyanov. Efficient Large Scale Language Modeling with Mixtures of Experts. *arXiv preprint arXiv:2112.10684*, 2021.

[11] Peter F. Brown, John Cocke, Stephen A. Della Pietra, Vincent J. Della Pietra, Fredrick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. A Statistical Approach to Machine Translation. *Computational Linguistics*, 16(2):79–85, 1990.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 2020.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, pages 579–594, 2018.

[15] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling Language Modeling with Pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[16] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, pages 613–627, 2017.

[17] Raj Dabre, Chenhui Chu, and Anoop Kunchukuttan. A Survey of Multilingual Neural Machine Translation. *ACM Computing Surveys*, 53(5), 2020.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[19] Ming Ding, Zhuoyi Yang, Wenyi Hong, Wendi Zheng, Chang Zhou, Da Yin, Junyang Lin, Xu Zou, Zhou Shao, Hongxia Yang, and Jie Tang. CogView: Mastering Text-to-Image Generation via Transformers. *Advances in Neural Information Processing Systems*, 2021.

[20] Lucas Dixon, John Li, Jeffrey Sorensen, Nithum Thain, and Lucy Vasserman. Measuring and Mitigating Unintended Bias in Text Classification. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 67–73, 2018.

[21] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathy Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. *arXiv preprint arXiv:2112.06905*, 2021.

[22] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. TurboTransformers: An Efficient GPU Serving System for Transformer Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[23] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low Latency RNN Inference with Cellular Batching. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[25] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[26] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training Compute-Optimal Large Language Models. *arXiv preprint arXiv:2203.15556*, 2022.

[27] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, pages 269–286, 2018.

[28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. *Advances in Neural Information Processing Systems*, 2019.

[29] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant Video Processing. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 29–42, 2018.

[30] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[31] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*, 2020.

[32] Daniel Khashabi, Sewon Min, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. UNIFIEDQA: Crossing Format Boundaries with a Single QA System. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1896–1907, 2020.

[33] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural Questions: a Benchmark for Question Answering Research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019.

[34] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and Parallel GPU Task

Scheduling for Deep Learning. *Advances in Neural Information Processing Systems*, 2020.

[35] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, pages 611–626, 2018.

[36] Gongzheng Li, Yadong Xi, Jingzhen Ding, Duan Wang, Bai Liu, Changjie Fan, Xiaoxi Mao, and Zeng Zhao. Easy and Efficient Transformer: Scalable Inference Solution For large NLP model. *arXiv preprint arXiv:2104.12470*, 2021.

[37] Opher Lieber, Or Sharir, Barak Lenz, and Yoav Shoham. Jurassic-1: Technical details and evaluation. 2021.

[38] Xudong Lin, Gedas Bertasius, Jue Wang, Shih-Fu Chang, Devi Parikh, and Lorenzo Torresani. Vx2text: End-to-end learning of video-based text generation from multimodal inputs. In *Proceedings of the 2021 IEEE Conference on Computer Vision and Pattern Recognition*, pages 7005–7015, 2021.

[39] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. *Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks*, pages 881–897. 2020.

[40] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a Suit of Armor Conduct Electricity? A New Dataset for Open Book Question Answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2381–2391, 2018.

[41] Ramesh Nallapati, Bowen Zhou, Cícero Nogueira dos Santos, Çaglar Gülçehre, and Bing Xiang. Abstractive Text Summarization using Sequence-to-sequence RNNs and Beyond. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*, pages 280–290, 2016.

[42] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. TensorFlow-Serving: Flexible, High-Performance ML Serving. *Workshop on Machine Learning Systems at NIPS 2017*, 2017.

[43] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, 2019.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 2019.

[45] Romain Paulus, Caiming Xiong, and Richard Socher. A Deep Reinforced Model for Abstractive Summarization. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

[46] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[47] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[48] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem

Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling Language Models: Methods, Analysis & Insights from Training Gopher. *arXiv preprint arXiv:2112.11446*, 2021.

[49] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[50] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. *arXiv preprint arXiv:2201.05596*, 2022.

[51] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-Shot Text-to-Image Generation. In *Proceedings of the 38th International Conference on Machine Learning*, pages 8821–8831, 2021.

[52] Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Eric Michael Smith, Y-Lan Boureau, and Jason Weston. Recipes for Building an Open-Domain Chatbot. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 300–325, 2021.

[53] Timo Schick and Hinrich Schütze. Exploiting Cloze-Questions for Few-Shot Text Classification and Natural Language Inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 255–269, 2021.

[54] Abigail See, Peter J. Liu, and Christopher D. Manning. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, 2017.

[55] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. *Advances in Neural Information Processing Systems*, 2018.

[56] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[57] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast Video Classification via Adaptive Cascading of Deep Models. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3646–3654, 2017.

[58] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[59] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990*, 2022.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. *Advances in Neural Information Processing Systems*, 2017.

[61] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. LightSeq: A High Performance Inference Library for Transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pages 113–120, 2021.

[62] Zihao Wang, Wei Liu, Qian He, Xinglong Wu, and Zili Yi. Clip-gen: Language-free training of a text-to-image generator with clip. *arXiv preprint arXiv:2203.00386*, 2022.

[63] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. Finetuned Language Models are Zero-Shot Learners. In *Proceedings of the 10th International Conference on Learning Representations*, 2022.

[64] Ronald J. Williams and David Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1(2):270–280, 1989.

[65] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, Attend and Tell: Neural Image

Caption Generation with Visual Attention. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 2048–2057, 2015.

[66] Zhilin Yang, Ye Yuan, Yuexin Wu, William W. Cohen, and Ruslan R. Salakhutdinov. Review Networks for Caption Generation. *Advances in Neural Information Processing Systems*, 2016.

[67] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models. *arXiv preprint arXiv:2205.01068*, 2022.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences

Mingcong Han[1,2], Hanze Zhang[1,4], Rong Chen[1,2], and Haibo Chen[1,3]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University [2]Shanghai AI Laboratory
[3]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
[4]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, China

Abstract

Many intelligent applications like autonomous driving and virtual reality require running both latency-critical and best-effort DNN inference tasks to achieve both real time and work conserving on GPU. However, commodity GPUs lack efficient preemptive scheduling support and state-of-the-art approaches either have to monopolize GPU or let the real-time tasks to wait for best-effort tasks to complete, which causes low utilization or high latency, or both.

This paper presents REEF, the first GPU-accelerated DNN inference serving system that enables microsecond-scale kernel preemption and controlled concurrent execution in GPU scheduling. REEF is novel in two ways. First, based on the observation that DNN inference kernels as mostly idempotent, REEF devises a reset-based preemption scheme that launches a real-time kernel on the GPU by proactively killing and restoring best-effort kernels at microsecond-scale. Second, since DNN inference kernels have varied parallelism and predictable latency, REEF proposes a dynamic kernel padding mechanism that dynamically pads the real-time kernel with appropriate best-effort kernels to fully utilize the GPU with negligible overhead. Evaluation using a new DNN inference serving benchmark (DISB) with diverse workloads and a real-world trace on an AMD GPU shows that REEF only incurs less than 2% overhead in the end-to-end latency for real-time tasks but increases the overall throughput by up to 7.7×, compared to dedicating the GPU to real-time tasks. To demonstrate the feasibility of our approaches on closed-source GPUs, we further ported and evaluated a restricted version of REEF on an NVIDIA GPU with a reduction of the preemption latency by up to 12.3× (from 6.3×).

1 Introduction

Deep Neural Network (DNN) inference has been widely adopted by modern intelligent applications, such as autonomous driving [2, 37, 41, 80], virtual reality [58, 83], speech/image recognition [32, 75], and healthcare [19, 24], just to name a few. Many of them demand real-time inference serving in mission-critical tasks, where GPUs have emerged as a popular accelerator to serve DNN inferences [15, 33, 47, 89].

Although the low-latency demand of DNN inferences can be fulfilled by dedicating the whole GPU to sequentially serve requests from a single DNN application [10, 80, 91], it is hard to fully exploit the massive parallelism of the GPU [47]. Hence, it is a common practice to share a GPU among multiple applications with different timing constraints in emerging intelligent systems [41, 78], which can greatly improve overall throughput, as shown in Fig. 1(a). For example, autonomous vehicles use DNNs to recognize obstacles and traffic lights [9, 59], which are latency-critical tasks (called *real-time* tasks in this paper). Meanwhile, other tasks with no hard real-time requirement [78] (called *best-effort* tasks in this paper), such as monitoring human driver's emotion and fatigue, are also served within the GPU using DNNs [19, 48, 84].

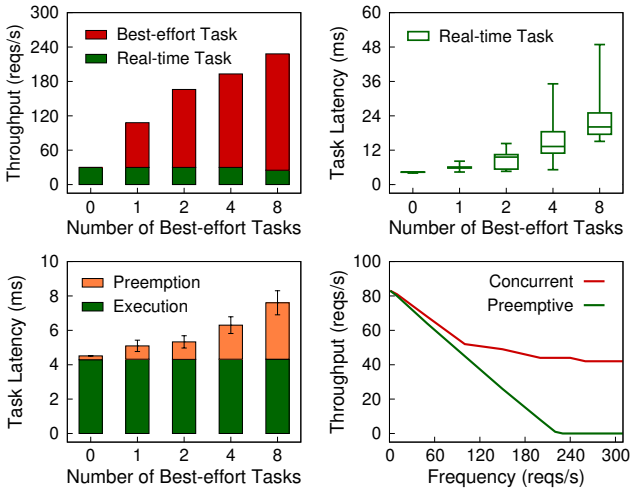


Fig. 1: *(a) The overall throughput of DNN inferences (both real-time and best-effort tasks) and (b) the end-to-end latency of real-time tasks when using concurrent GPU scheduling (i.e., multiple GPU streams [46, 49, 60]), (c) the end-to-end latency of real-time tasks when using preemptive GPU scheduling (i.e., wait-based preemption [12, 77, 90]), and (d) the throughput of best-effort tasks as the frequency of real-time tasks increases.* **Workload**: *VGG [68] (real-time) and ResNet [30] (best-effort).* **Testbed**: *one AMD Radeon Instinct MI50 GPU with 16 GB of memory (see §7 for details).*

Typically, DNN inferences have two potentially conflicting goals for GPU scheduling. First, the real-time tasks should be treated as first-class citizens on the GPU without interference from other tasks to achieve *low end-to-end latency*. Second, both real-time tasks and best-effort tasks should be served concurrently on the GPU to achieve high overall throughput (*work-conserving*).

State-of-the-art GPU libraries (e.g., CUDA [52] and ROCm [3]) commonly provide multiple GPU streams (e.g., CUDA Streams [60]) to concurrently execute multiple tasks on the same GPU. However, as shown in Fig. 1(b), although the end-to-end inference latency of real-time tasks is low (about 4 ms) and stable when monopolizing the GPU, the tail latency of real-time tasks significantly increases by over an order of magnitude (close to 50 ms) when running concurrently with best-effort tasks. This, unfortunately, is unacceptable for real-time scenarios [85].

Similar to operating systems using preemptive scheduling to provide real-time guarantees, an intuitive approach is to provide preemption for GPU scheduling, which is unfortunately missing in commodity GPUs [70]. Prior work [12, 77, 90] proposed a wait-based approach to passively waiting until the completion of running blocks, which may cause a preemption delay of several milliseconds. Although it may be sufficient for traditional GPU workloads, this approach is still far from optimal for DNN inference tasks since the preemption latency is non-trivial compared to the execution time of real-time inference tasks, as shown in Fig. 1(c). Further, when the real-time inference requests arrive at a high frequency (e.g., camera (120 reqs/s) [16] or multiple sensors [41]), the best-effort tasks may even get starved, as shown in Fig. 1(d).

This paper presents REEF, the first DNN inference serving system for commodity GPUs with microsecond-scale kernel preemption and controlled concurrent execution in GPU scheduling to achieve both *real time* and *work conserving*. Specifically, the arriving real-time task should instantly preempt the GPU from the running best-effort kernels without waiting for their completion. Meanwhile, the best-effort kernels should be executed concurrently by using GPU resources leftover from the real-time kernels.

A key insight of REEF is that each kernel in DNN inference is mostly *idempotent*. This implies that the running best-effort kernels can be proactively killed and restored without saving contexts. Based on this, REEF proposes a *reset-based preemption* scheme. To thoroughly flush hundreds of outstanding kernels in both GPU runtime and devices, REEF designs different approaches to resetting different software queues and retrofits the GPU driver to exactly use existing hardware mechanisms to reset compute units while preserving device memory of the GPU. It can improve both kernel preemption and restore. Therefore, REEF can launch a real-time task on the GPU in tens of microseconds, regardless of the number of preempted kernels and their execution time.

REEF further proposes a *dynamic kernel padding* mechanism based on the observation that the execution time of GPU kernels in DNN inferences is deterministic and *predictable*. This implies that the pending best-effort kernels can be carefully selected to pad the real-time kernel without performance interference, based on offline profiling in advance. REEF extended GPU compiler to construct a template of padded kernels by using function pointers. Further, to eliminate the overhead of indirect function calls on the GPU, REEF introduces proxy kernels to address register allocation problem and avoid unnecessary context saving at runtime. Therefore, REEF can concurrently execute the real-time task with best-effort tasks at the expense of negligible performance and memory overhead (less than 1% and about 10 KB).

We have implemented **REEF** by extending Apache TVM [73] (a compiler for deep learning) and AMD ROCm [3] (an open-source GPU computing platform). We evaluate REEF using a new DNN Inference Serving Benchmark (DISB) with diverse workloads and models, as well as a real-world trace from Apollo [7] (an open autonomous driving platform). Our experimental results show that REEF only incurs less than 2% of the end-to-end latency overhead for real-time tasks but increases overall throughput by up to 4.3×, compared to dedicating the GPU to real-time tasks. Our approach further reduces the preemption latency by over one order of magnitude against the state-of-the-art, less than 40 microseconds for all models. To demonstrate the feasibility of our approaches on closed-source GPUs, we further ported and evaluated a restricted version of REEF on an NVIDIA GPU with a reduction of the preemption latency by up to 12.3× (from 6.3×).

Contributions. We summarize our contributions as follows.

- An in-depth understanding on the characteristics of GPU-accelerated DNN inferences such as idempotence and the issues of state-of-the-art GPU scheduling schemes (§2).
- A new reset-based preemption scheme that can launch a real-time kernel on the GPU in a few microseconds, regardless of the number of preempted kernels (§4).
- An elegant mechanism that can dynamically pad the real-time kernel with best-effort kernels to fully exploit the massive parallelism of the GPU (§5).
- An implementation (§6) on both AMD and NVIDIA GPUs and an evaluation that demonstrates the advantage and efficacy of REEF over state-of-the-art (§7).

The source code of REEF is publicly available at https://github.com/SJTU-IPADS/reef. The DNN Inference Serving Benchmark (DISB) framework can be obtained separately from https://github.com/SJTU-IPADS/disb.

2 Background and Motivation

2.1 Characterizing GPU-Accelerated DNN Inference

Deep neural network (DNN) comprises multiple instances of versatile layers, such as convolutional, pooling and fully-connected layers. GPUs have been widely exploited to accelerate DNN inference serving [20, 28, 64]. To serve inference requests on GPUs, the pre-trained DNN model (e.g., ResNet [30]) is loaded into GPU memory ahead of time. Fig. 2 outlines the implementation of GPU-accelerated DNN inference. For each arriving request, all kernels of the DNN

```
# device codes
__global__ void conv_relu(in, weight, out):
1   sum = 0;
2   for i in range(0,3)
3      for j in range(0,3)
4         sum += in[..] × weight[..]
5   out[..] = ReLU(sum)
__global__ void dense(in, weight, bias, out):
6   sum = 0;
7   for i in range(0,512)
8      sum += in[..] × weight[..]
9   out[..] = sum + bias[..]

# host codes
void inference(...):
10  memcpyH2D(in, in_host, in_sz)  # copy in to GPU
11  conv_relu <<<dim(32), ..>>> (in, .., buf_conv)
12  ...  # launch other kernels
13  pooling <<<dim(64), ..>>> (.., buf_pool)
14  dense <<<dim(10), ..>>> (buf_pool, .., buf_dense)
15  softmax <<<dim(1), ..>>> (buf_dense, .., out)
16  memcpyD2H(out_host, out, out_sz)  # copy out to CPU
```

Fig. 2: *An example of DNN inference using a model like ResNet.*

model are executed in turn with the input, and the resulting output is returned to the DNN application.

DNN inference is now used by both *real-time* (RT) tasks, such as obstacle and traffic lights recognition [9, 59], and *best-effort* (BE) tasks, such as emotion and fatigue monitoring [19, 48, 84]. The real-time tasks are latency-critical, because violating the end-to-end latency requirement may cause system failures or even safety problems. In addition, such requests are usually issued periodically at various frequencies by input sensors (e.g., camera and LiDAR [7, 41]). On the contrary, the best-effort tasks have no hard timing requirement, but are repetitively executed in the background.

Idempotence. The GPU-accelerated DNN model for inference tasks consists of a sequence of kernels, which implement one or several DNN layers. We observe that GPU kernels in DNN models are mostly *idempotent* as they consist of almost only dense linear algebra computations without side effects.[1] Hence, the kernel can always produce the same output with the same input no matter it has been retried or not. Meanwhile, in the DNN model, the (k)-th kernel always uses the outputs of the (k-1)-th kernel and static arguments (e.g., weight) as inputs, e.g., `conv_relu` and `dense` kernel in Fig. 2. Therefore, the execution of DNN inference task can be restored from any kernel before the interrupted kernel and will not change the inference results.

Massive kernels. Unlike traditional GPU applications that only contain a few kernels (e.g., at most 14 kernels in Rodinia [11]), it is common to see hundreds of kernels in modern DNN models (see Table 1). In response, large amounts of kernels—usually hundreds or more—would be submitted in advance to hide the lengthy kernel launching time. Further, to fully exploit the GPU, the serving system may concurrently execute multiple kernels from different inference tasks using the same or different DNN models. Therefore, the performance penalty of preempting the GPU would be significant (a few milliseconds) and even comparable to the execution time of hundreds of kernels.

[1] We validated using our tool that all 320 GPU kernels of the 11 DNN models from Apache TVM's test suite [72] are idempotent.

Table 1: *The amount of GPU kernels in DNN models evaluated in §7 and the execution time (in millisecond). The codes are generated by TVM [15] and run on AMD Radeon Instinct MI50 GPU.*

Model	ResNet	DenseNet	VGG	Inception	Bert
#Kernels	307	207	55	146	205
Exec. Time	13.6	3.5	4.4	8.3	5.4

Latency predictability. We observe that the execution time of GPU kernels in DNN inferences is deterministic and predictable when running individually on the GPU (no interference). The reasons are two-fold. First, the kernel is mostly linear algebra computations such as matrix multiplication and convolution, which contains neither conditional branches nor inconstant loops. Second, all kernel arguments (e.g., input and weights) and the output are fixed-size arrays. Therefore, the execution time of such kernels is independent of the input of inference request and can be measured and accurately predicted in advance. In practice, we observe that the variance in kernel execution time of DNN models is typically only a few microseconds (see Fig. 3(a)). This is also confirmed in recent literature [6, 28, 47].

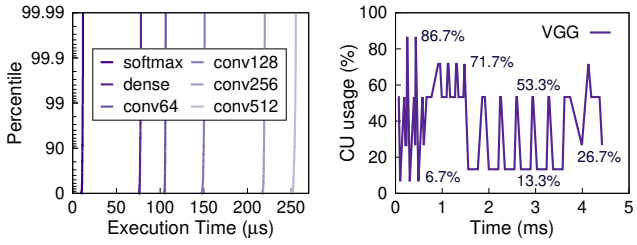


Fig. 3: *(a) The CDF of execution time of several typical kernels in VGG, and (b) the timeline of CU usage during VGG execution on a GPU with 60 CUs. Note that the execution time of GPU kernels in VGG covers a fairly wide range from 10 µs to 255 µs (see Fig. 10).*

Varied parallelism. The GPU kernels in DNN inferences usually exhibit completely different parallelism due to varied input scales. For example, as shown in Fig. 2, the `pooling` kernel uses 64 thread blocks, while the `softmax` kernel just uses 1 thread block. Consequently, the computational demand for DNN inferences, namely the number of compute units (CUs), is ever-changing during the execution. As an example, Fig. 3(b) shows the CU usage during VGG execution varies between 6.7% and 86.7%. Therefore, to efficiently exploit the GPU, it is indispensable to leverage a dynamic mechanism to select and execute multiple kernels from different DNN inference tasks at runtime.

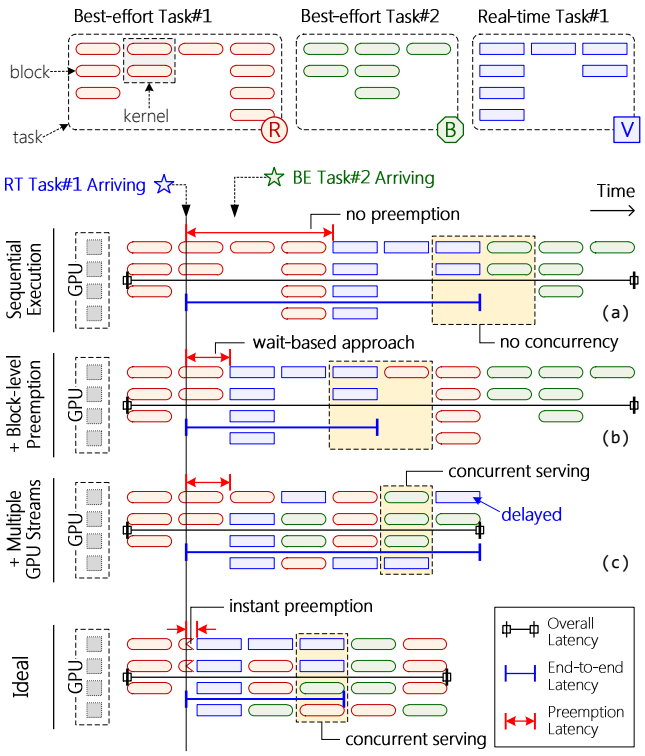


Fig. 4: *An example of GPU task scheduling with different kernel preemption and parallelism schemes for a hybrid workload, which contains two best-effort and one real-time DNN inference tasks. The GPU has four compute units (CUs).*

2.2 State-of-the-art GPU Scheduling

As stated before, DNN inference serving system relies on GPU scheduling to meet two potentially conflicting performance goals: low latency and work conserving. Although GPU scheduling has been widely studied in the HPC community [1, 8, 12, 13, 27, 43, 77, 82, 88], the unique characteristics of DNN inferences and the two performance goals introduce new challenges for GPU scheduling. We review the state-of-the-art schemes of GPU scheduling and discuss the performance issues when serving various DNN inferences through a brief example, as shown in Fig. 4.

Sequential execution. Most existing DNN serving systems, such as Clockwork [28], use sequential execution to avoid interferences among tasks. Thus, each task can achieve optimal execution latency, as shown in Fig. 4(a). However, the end-to-end latency of RT tasks might be significantly extended due to lengthy preemption latency (red dimension line), since it has to wait for the completion of previous tasks (*no preemption*). Further, this scheme has a poor overall throughput, due to sequentially serving inference tasks (i.e., *no concurrency*).

Block-level preemption. To reduce end-to-end latency for real-time tasks, it is necessary to preempt the GPU from running best-effort tasks. However, it is difficult to implement preemptive scheduling on the GPU due to the large context (e.g., a large amount of registers) [56, 70]. Meanwhile, commodity GPUs also lack hardware support for the preemption mechanism.[2] As a compromise, prior work [8, 90] proposes wait-based approaches to implementing block-level preemption for GPU scheduling. The real-time task still needs to passively wait until the completion of running blocks, as shown in Fig. 4(b). Further, the preemption latency will increase with the number of preempted kernels (see Fig. 1(c)). As a compromise, prior work [8, 90] has to limit the number of kernels submitted to the GPU, which is impractical for DNN inferences. Further, a high-frequency real-time task will break the execution of best-effort tasks, even leading to starvation (see Fig. 1(d)).

Multiple GPU streams. To improve overall throughput, modern GPU libraries (e.g., CUDA [52] and ROCm [3]) commonly provide multiple GPU streams (e.g., CUDA Streams [60]) to concurrently execute kernels from independent tasks. The runtime scheduler dispatches kernels from GPU streams on demand to keep all compute units (CUs) busy, as shown in Fig. 4(c). Although leveraging multiple GPU streams can improve throughput (see Fig. 1(a)), the latency of real-time tasks can be significantly degraded by concurrent tasks, e.g., the last kernel of RT Task#1 in Fig. 4(c). Even worse, the latency overhead will increase with the number of concurrent tasks (see Fig. 1(b)).

3 REEF Overview

3.1 System Architecture

The goal of REEF is to provide preemptive GPU scheduling to achieve real time for latency-critical tasks and work conserving for best-effort tasks (see ideal scheduling for the example in Fig. 4). Based on the insight that DNN inference kernels are mostly idempotent and there are a massive number of kernels with varied parallelism and predictable latency, REEF provides two novel designs called reset-based preemption and dynamic kernel padding.

Fig. 5 illustrates an overview of REEF's architecture. REEF consists of (a) an offline part, which compiles and loads user-provided DNN models, and (b) an online part, which schedules and serves DNN inference requests.

DNN model preparation (offline). Typically, DNN models are first compiled and optimized for accelerator back-ends (e.g., GPU) and then loaded into the model pool. Inspired by prior work [12, 36, 77], REEF extends the model compiler (e.g., TVM [15]) with a *code transformer* module, which first validates the idempotence of kernels in DNN models and then transforms the source code to assist GPU scheduling in REEF. Moreover, REEF develops a *kernel profiler* to measure the computational requirements and the execution time for each kernel of the model, which is accurate and practical for DNN models (see §2).

[2] Although NVIDIA claims that their GPUs have been equipped with preemption support since Pascal architecture [51], there is no public available information or a software controllable interface [12, 39, 77].

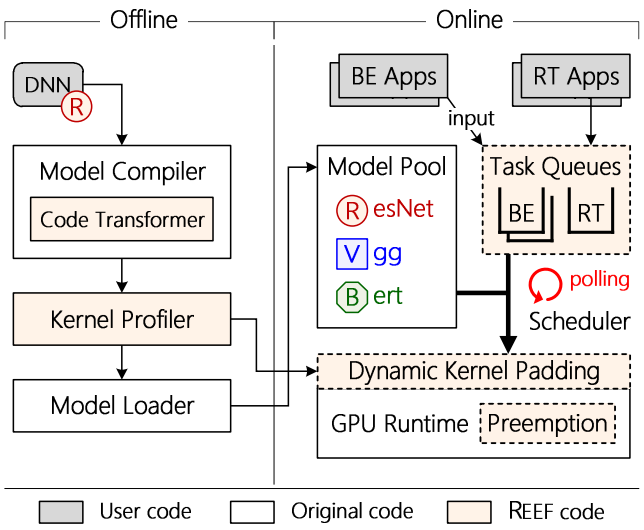


Fig. 5: *Architecture of* REEF. *Modules in boxes with dashed border are on the critical path of serving DNN inference requests. Other modules do not directly impact serving latency and throughput.*

DNN inference serving (online). REEF extends a state-of-the-art GPU runtime (e.g., ROCm [3]) with four major components for DNN inference serving.

Task Queues. REEF maintains one real-time task queue and several best-effort task queues. Each queue is bound to a GPU stream for launching GPU kernels, where inference requests are served in a FIFO order. For simplicity, REEF executes real-time requests one at a time. Note that any scheduling policy that treats the whole GPU as a single device, such as EDF [10], can be adopted by REEF for real-time requests. Further, REEF offers an RPC-based interface for DNN-based applications to deliver inference requests to task queues.

Scheduler. The scheduler in REEF uses busy polling on task queues and assigns tasks to the associated GPU streams. Corresponding to whether there are real-time tasks, REEF provides two execution modes, namely *real-time* mode and *normal* mode. The scheduler will switch from normal mode to real-time mode when encountering real-time tasks, and switch back to normal mode when the real-time task queue is empty.

Preemption module. In normal mode, REEF concurrently serves best-effort tasks from different task queues using multiple GPU streams [3, 60] provided by GPU runtime. In real-time mode, REEF first uses the preemption module to instantly preempt the GPU from all running best-effort tasks (§4) and then launches the real-time task on the GPU immediately.

Dynamic kernel padding (DKP). In real-time mode, before launching a real-time kernel, the DKP module will select appropriate best-effort kernels and dynamically pad them to the real-time kernel (§5). REEF will execute the padded kernel on the GPU to achieve high throughput. Note that the best-effort kernels will only use GPU resources leftover from the real-time kernel.

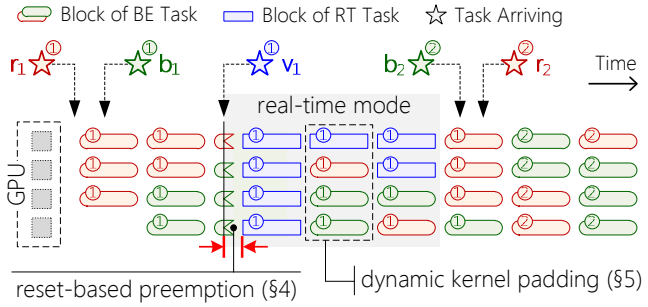


Fig. 6: *An example of timeline in* REEF. *The DNN inference tasks here are the same as that in Fig. 4.*

3.2 An Illustrative Example

Fig. 6 illustrates the timeline of scheduling five DNN inference tasks in REEF. Upon receiving the first two best-effort requests r_1 and b_1, REEF runs in normal mode, and the kernels of two different tasks are scheduled to two different GPU streams. The GPU runtime will concurrently execute the kernels on the GPU. While r_1 and b_1 execute, a real-time request v_1 arrives. The scheduler immediately switches to real-time mode, and GPU runtime instantly preempts the GPU by killing all running kernels of best-effort tasks (i.e., r_1 and b_1). Meanwhile, the DKP module selects appropriate kernels from restored tasks to dynamically pad the kernels of real-time task v_1. After that, the padded kernel will be executed on the GPU alone. While v_1 is completed, the scheduler switches back to normal mode. All running and later best-effort tasks (i.e., r_1, b_1, b_2, and r_2) will concurrently execute on the GPU through two GPU streams.

4 Reset-based Preemption

The key insight behind our idea, namely reset-based preemption, is that the GPU kernels in DNN models are mostly *idempotent*, which enables *proactive* preemption—killing all running kernels on the GPU immediately and restoring them later. The benefits are two-fold. First, it avoids saving and restoring the large context of the GPU (e.g., a 256 KB register file per CU) [70]. Second, there is no need to wait for all running kernels to complete, which can take hundreds of microseconds.

However, there are still new challenges before making our reset-based preemption come true on commodity GPUs. Except for the kernels running on the GPU, hundreds of launched kernels are buffered in multiple queues maintained by GPU runtime. This is necessary to hide the kernel launch time and fully exploit the massive parallelism of GPU. Whereas, evicting all launched kernels makes it indeed difficult to preempt the GPU in tens of microseconds.

Fig. 7 illustrates the lifetime of launched kernels in the GPU runtime and devices. First, the scheduler launches all kernels of an inference task and specifies a GPU stream for each task. The GPU runtime maintains a linked list, called

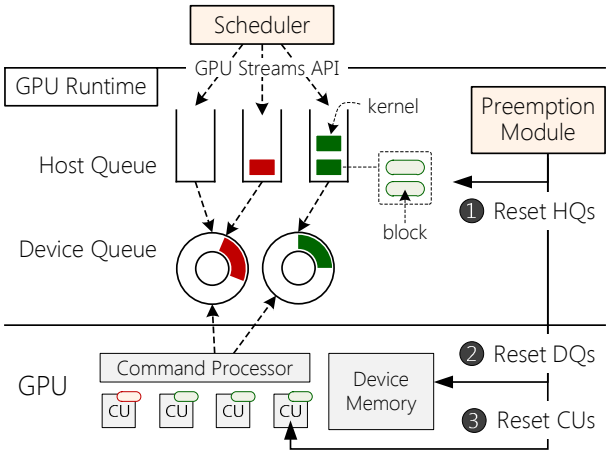


Fig. 7: *Extended GPU runtime in* REEF *for instant preemption.*

host queue, for each GPU stream to buffer launched kernels. Each host queue has a background thread that transmits the buffered kernels asynchronously to a ring buffer, called *device queue*, which is accessed by CPU and GPU simultaneously. The command processor of GPU will poll all device queues to fetch the buffered kernels and eventually dispatches them to *compute units*. Therefore, launched kernels of an inference task may exist in three places, namely host queues (HQs), device queues (DQs), and compute units (CUs). To achieve instant preemption, kernels in all three places must be evicted.

4.1 Evicting Buffered Kernels

The reset-based approach requires proactively evicting all buffered kernels from both host queues and device queues. For host queues, it is straightforward to reset them (❶ in Fig. 7), dequeuing all buffered kernels and reclaiming memory, as they are fully controlled by the GPU runtime. For device queues, however, the GPU runtime cannot evict buffered kernels from device queues, because the command processor of GPU can directly fetch kernels from device queues [23], resulting in data races and unpredictable results. In addition, the CPU also does not provide a way to safely evict kernels from device queues. A potential solution is to notify the GPU to re-register a new device queue [62]. However, it would incur an unacceptable latency overhead (e.g., about 1 ms on our testbed).

Inspired by evictable kernels [12], we propose *lazy eviction* to reset device queues without extending GPU runtime and hardware. The code transformer of REEF injects a piece of code at the beginning of each kernel in advance, which checks the *preemption flag* to realize whether it has been evicted. When the preemption flag is true, the kernel will voluntarily terminate itself. Therefore, when a preemption occurs, the preemption module will immediately set the preemption flag to true in GPU memory (see ❷ in Fig. 7). The kernels buffered in device queues will be fetched and dispatched to the CUs as usual, but will terminate themselves immediately.

Our initial queue eviction mechanism imposes a non-trivial overhead on the preemption process, taking more than 500 µs to preempt a single task (see §7.3). An in-depth analysis shows that the overhead comes mainly from (a) reclaiming memory from the host queue and (b) waiting to fetch kernels from the device queue. Therefore, we propose two optimizations to mitigate overheads.

Asynchronous memory reclamation. The preemption latency is proportional to the host queue length when using synchronous memory reclamation for evicted kernels in the host queues. Therefore, the performance penalty of preempting a DNN inference task would be significant, since it requires buffering hundreds of kernels in the host queue. To instantly evict GPU kernels from the host queue, REEF leverages a background GC thread to reclaim memory asynchronously. Specifically, REEF resets the host queue by simply nullifying the head pointer first and then notifying the GC thread to reclaim memory in the background.

Device queue capacity restriction. Although using lazy eviction can terminate kernels in the device queue immediately at the beginning of execution, the kernels still have to be fetched and dispatched to the CU, which takes around 20 µs per kernel. It is common to buffer hundreds of kernels in a device queue, since it can reduce the frequency of context switches by filling up the device queue with a large number of kernels from host queues at a time. However, it may also increase the preemption delay to even more than 1 ms. Therefore, REEF restricts the capacity of the device queue to achieve microsecond-scale kernel preemption. Tuning the device queue capacity provides a tradeoff between preemption latency and execution time. As the queue capacity decreases, the preemption latency also decreases because fewer kernels need to be evicted, but normal execution time increases because the GPU has more idle time waiting for the runtime to fill device queues with the kernels from host queues. We empirically choose a device queue capacity to 4 on our testbed, since it is sufficient to reset the device queue in 30 µs with negligible overhead on normal execution time (i.e., less than 0.3%). Furthermore, using a smaller device queue also produces slightly higher CPU utilization (e.g., about 15% increase) due to more frequent filling of the device queue.

4.2 Killing Running Kernels

To avoid waiting for the completion of running kernels, the reset-based preemption proactively kills the running kernels in the GPU. Unfortunately, there is neither an API provided by GPU runtime nor a functionality exposed by GPU driver that can kill the running kernels from the host side. We observed that GPU driver has the ability to terminate CPU process and also kill associated GPU kernels, even when the kernel stucks in an infinite loop. It implies that GPU driver can indeed kill an uncompleted kernel. However, this function will also reclaim GPU memory allocated by the process and GPU kernels. Thus, the preempted kernel has to reload DNN model parameters to GPU memory, taking even a few seconds.

To remedy it, REEF retrofits the kernel killing function of GPU driver and exposes it to the preemption module in GPU runtime. The new function will instruct the command processor to kill all running kernels on the CUs but preserve their running state in GPU memory. The preemption module will use it to kill all running kernels (see ❸ in Fig. 7) after evicting host queues and device queues.

4.3 Restoring Preempted Tasks

The best-effort tasks should be restored after being preempted. In general, the task has to be re-executed from the beginning, and is assumed to have no side effects. Fortunately, the *idempotence* characteristic of kernels in the DNN model ensures that the execution of DNN inference task can be restored from any kernel before the interrupted kernel. This implies that the scheduler can safely re-execute the preempted best-effort tasks. However, this may incur severe additional overhead because DNN models commonly have massive kernels (usually hundreds or more). Therefore, it is important to restore the preempted task from the kernel close to where it was interrupted. Unfortunately, it is almost impossible to precisely identify the interrupted kernel, because the kernel running on the CUs is killed directly by the command processor of GPU.

To remedy this problem, REEF adopts an *approximation* approach to ensure that the preempted task is restored from at most a *constant* number (c) of kernels before the interrupted kernel. More specifically, the preemption module first records the last kernel (k_l) transmitted to the device queue when it starts resetting the task queue, and then restores the preempted task from c kernels before k_l, where c denotes the device queue capacity. We observe that the command processor sequentially fetches a kernel from the device queue and runs it on the CUs. This implies that the interrupted kernel will not be earlier than c kernels before the last kernel (k_l) in the device queue. Furthermore, REEF will redundantly execute at most c+1 kernels. Since c is configured to be relatively small (i.e., 4), the restore overhead is negligible (about 30 µs).

4.4 Preemption on closed-source GPUs

Many commodity GPUs (e.g., NVIDIA GPUs) are still closed source. This poses new challenges to our reset-based preemption scheme, which has to treat the GPU runtime as a black box. The primary limitation is that we cannot reset CUs to proactively kill running kernels (❸ in Fig. 7). Apart from that, REEF is also unable to manipulate host queues and device queues directly outside of the GPU runtime. But fortunately, the lazy eviction scheme proposed by REEF for resetting DQs (❷ in Fig. 7) does not require any modification to the GPU runtime.

We propose a restricted version of reset-based preemption, called REEF-N, for closed-source GPUs. REEF-N first wraps each GPU stream, the general abstraction provided by GPU runtime, into a *virtual* host queue (vHQs), which intercepts and buffers all launched kernels. Similar to the (physical) HQ inside the GPU runtime, each vHQ also has a background thread to transmit buffered kernels asynchronously to the GPU runtime. After that, REEF-N treats the whole GPU runtime as several device queues (one for each GPU stream), such that REEF can easily reset vHQs to evict buffered kernels, instead of resetting HQs directly (❶ in Fig. 7). REEF-N still follows the lazy eviction to reset DQs, and then waits for all running kernels to complete. Finally, to simulate DQ capacity restriction, REEF limits the number of outstanding kernels in the GPU runtime; the background thread of vHQ transmits a fixed number of kernels to the GPU runtime in a closed loop.

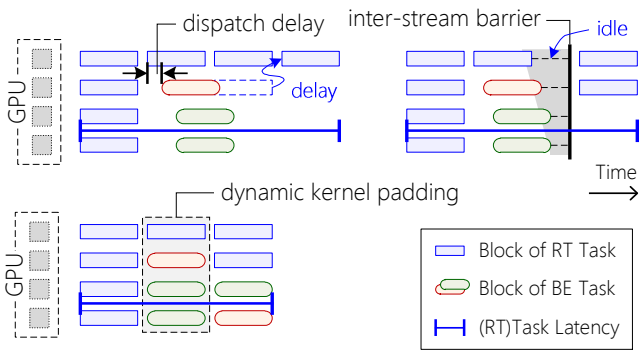


Fig. 8: *An example of serving multiple kernels in parallel with different approaches.*

5 Dynamic Kernel Padding

To achieve high throughput, both real-time and best-effort tasks should be concurrently executed on the GPU to achieve work conserving. However, to avoid interference with real-time tasks, the best-effort tasks should be only served by using GPU resources leftover from the real-time tasks. Regrettably, none of the existing approaches can provide such controlled concurrent execution on the GPU.

First, using different GPU streams to launch real-time and best-effort tasks cannot avoid interfering with each other. As shown in Fig. 8, the dispatch delay between GPU streams (20–40 µs) might postpone the execution of real-time kernels or limit the available resources (e.g., CUs) to them. Using additional inter-stream barriers to synchronize kernel dispatch among CUs will also cause performance overhead.

Second, static kernel fusion [74] can merge multiple kernels from different tasks into a single one at compile time and then launch the fused kernel on the GPU using a single stream. It can avoid interference between real-time tasks and best-effort tasks in advance. However, static kernel fusion has to pre-compile all possible combinations of all kernels in DNN models to enable scheduling at runtime. As mentioned above, DNN inferences have hundreds of kernels in common (see Table 1), which makes it impractical for static kernel fusion. For example, it requires more than 35 GB of GPU memory to store the fused kernels for five DNN models in Table 1—considering only all combinations of no more than three kernels.

```
# device codes
__device__ void dense(in, weight, bias, out): ...

__global__ void dkp(rt_kern, rt_args,
                    be_kerns, be_argss):
1   ncus = rt_kern.ncus  # number of CUs
2   if (cu_id() < ncus) then
3      rt_kern(rt_args)  # run RT/kernel
4   else
5      ncus += be_kerns[i=0].ncus
6      while (cu_id() >= ncus)
7         ncus += be_kerns[++i].ncus
8      be_kerns[i](be_argss[i])  # run BE/kernel

# host codes
void inference(...):
    # set the real-time kernel w/ its args (e.g., dense)
9   rt_kern, rt_args = ...
    # select a set of best-effort kernels w/ their args
10  be_kerns, be_argss = kern_select(rt_kern)
11  dkp <<<..>>> (rt_kern, rt_args, be_kerns, be_argss)
12  ... # launch other dynamic padded kernels
```

Fig. 9: *Pseudocode for dynamic kernel padding in* REEF.

Our approach: dynamic kernel padding. Inspired by kernel fusion, our approach also combines real-time kernels and best-effort kernels into a single one and launches it using a single GPU stream, as shown in Fig. 8. Differently, we construct a template (called *dkp kernel*) at compile time and use *function pointer* to fill and execute kernels at runtime. Further, we dynamically select best-effort kernels to avoid interference with the real-time kernel.

Fig. 9 shows an example of a dkp kernel (`dkp`) for dynamic kernel padding, declared as a *global* function (i.e., kernel entry). Instead of being statically inlined into the dkp kernel, candidate kernel functions (e.g., `dense`) are declared as individual *device* functions, which can be passed as dkp kernel arguments and called by function pointers (line 3 and 8). The dkp kernel partitions the CUs to execute one real-time candidate kernel (`rt_kern`) and a set of best-effort candidate kernels (`be_kerns`) in parallel. It first allocates sufficient CUs for the real-time kernel (line 1–3) and then assigns the leftover CUs to the best-effort kernels (line 5–8). When launching a real-time kernel, the DKP module selects appropriate best-effort kernels to concurrently execute with the real-time kernel (line 10, see also §5.2).

5.1 Efficient Function Pointers

Without specific optimizations, the naive design would significantly decrease the performance of real-time kernels, due to the unique characteristics of *function pointers* on the GPU. We summarize the two key performance issues of the default function pointer mechanism on the GPU.

Limited register allocation. Unlike CPU programs, GPU programs require a diverse yet fixed amount of registers, which is counted at compile time and encoded into the model executable. Such an attribute prohibits the direct use of function pointers in GPU kernels, as the number of registers used by the indirectly called function cannot be determined statically. The default behavior of the GPU compiler is to assign a predefined static upper bound to limit the callee's register usage, which may force the callee to save variables on the stack due to the insufficient registers, leading to poor performance compared to purely using registers [43].

Expensive context saving. Indirect function calls on GPUs are much more expensive than CPU programs, due to the enormous context (e.g., dozens of registers) that needs to be saved and restored before and after the function call. For thousands of threads, there might be MB-sized registers saved and restored, introducing significant overheads. Although the compiler will inline as many functions as possible to avoid this overhead, indirect function calls via function pointers cannot be inlined, which may impose significant performance penalty on dynamic kernel padding.

REEF tackles the two above issues by introducing *global function pointer* as a substitution of the default function pointer mechanism. Since global functions are treated as kernel entries, the compiler neither applies register limitations nor adds context saving/restoring code to them. Thus, declaring candidate kernels as global functions instead of device functions can solve both issues. According to our observation, context saving in candidate kernels is actually unnecessary, as the dkp kernel exits immediately after calling `rt_kern` or `be_kerns[i]` (see Fig. 9). Therefore, the lack of context saving code in candidate kernels does not affect the execution correctness.

However, as the kernel entry, a global function cannot be called by another global function (e.g., dkp kernel). To bypass this restriction, we replace indirect function calls with jump instructions in assembly code, and manually prepare the initial state of candidate kernels by following the conventions [45]. This approach makes no changes to the compiler and only incurs a trivial function call overhead (around 1%).

Dynamic register allocation. The real-time kernel performance is still not ideal after applying the global function pointer technique because of the *over-allocation* problem. To meet the varied register demands of candidate kernels, the dkp kernel has to allocate as many registers as possible (i.e., over-allocation), which may decrease the CU occupancy[3], and thus increase the execution time. An intuitive solution is to overwrite the register count of the dkp kernel just-in-time before it is launched, making it adaptive to selected candidate kernels. Unfortunately, the kernel's register count has been loaded to the GPU memory with the model in the off-line phase (§3), which means overwriting its value requires a CPU-to-GPU memory copy before every kernel execution, severely affecting the execution performance.

REEF addresses the dynamic register allocation problem

[3]The CU occupancy implies how many blocks can be executed on a CU simultaneously. It depends on how many resources (e.g., register) each block demands. Higher CU occupancy can lead to better performance.

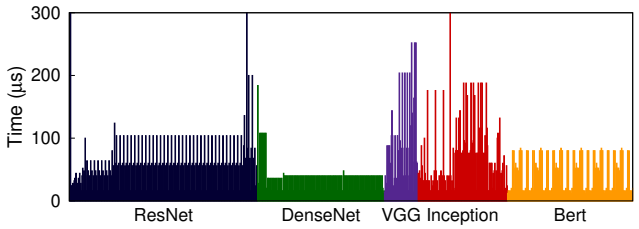


Fig. 10: *The measured execution time of kernels in five DNN models. The details of DNN models can be found in Table 1.*

by introducing a set of *proxy kernels*. Proxy kernels share the same source code as the dkp kernel in Fig. 9, but allocate different number of registers, allowing the scheduler to dynamically pick the proper proxy kernel according to each candidate kernel's register demand. Unfortunately, generating proxy kernels for every possible register count faces the kernel amount explosion problem. For example, on AMD Instinct MI50 GPU with at most 128 scalar registers and 256 vector registers for each thread, it will generate 32,768 proxy kernels to cover all possible register configurations.

To reduce the proxy kernel amount, we generate proxy kernels to cover all possible CU occupancies rather than register counts. Since proxy kernels are introduced to prevent over-allocation from decreasing the CU occupancy, proxy kernels that have different register count yet share the same CU occupancy are actually redundant and can be merged together. More specifically, there are 10 CU occupancy levels on AMD Instinct MI50 GPU we use, corresponding to 10 register count ranges, which allows us to generate only 10 proxy kernels, each allocating the maximum amount of registers allowed in a CU occupancy level. For each candidate kernel, the scheduler picks the proxy kernel with the fewest allocated registers that fulfill the candidate kernel's demand, which achieves the highest CU occupancy possible. This way, the amount of proxy kernels is narrowed down from 32,768 to 10 without affecting the candidate kernel's performance.

Dynamic shared memory. In addition to registers, over-allocation of shared memory may also decrease the CU occupancy of proxy kernels. Fortunately, the kernel is enabled to dynamically allocate shared memory by setting a property (i.e., "dynamic shared memory") when launching the kernel. During model compilation, REEF converts the declaration of variables from *fixed-size* shared memory to *dynamic* shared memory (i.e., adding $extern$ before $\_\_shared\_\_$). Consequently, the amount of shared memory used by proxy kernels can be set at runtime, depending on the maximum demand of candidate kernels.

5.2 Kernel Selection

For dynamic kernel padding, the *kernel selection policy* is important to avoid latency interference with real-time tasks, which selects a set of blocks from candidate best-effort kernels to share the GPU with the arriving real-time kernel. REEF proposes a greedy heuristic to ensure that the best-effort blocks will only use GPU resources (i.e., CUs) leftover from the real-time kernel. Specifically, it first reserves enough CUs for the real-time kernel, and then checks best-effort task queues to select appropriate blocks for the remaining CUs, until there are no free CUs or candidate tasks. The selected best-effort blocks should meet the following two rules.

Rule 1. The execution time of best-effort kernels must be shorter than that of the real-time kernel, since the execution time of the dkp kernel is determined by the slowest block. Based on the observation of latency predictability for GPU kernels in DNN models (see §2.1), we develop an offline kernel profiler to measure the computational requirements and the execution time for each kernels of loaded models.

Rule 2. The CU occupancy of best-effort kernels must be higher than that of the real-time kernel, since the CU occupancy of the dkp kernel is determined by the minimum of kernels. Note that the CU occupancy of kernels can be directly obtained from the source code of DNN models.

The kernel selection policy fully meets the design goal of treating the real-time tasks as first-class citizens on the GPU. It is not only efficient, selecting best-effort kernels in less than 1 µs, but also effective, limiting the latency overhead of real-time kernels to less than 1% on average, see §7.4 for details. However, the policy is also conservative, so the constraint may limit room for improvement in overall throughput. For example, when the execution time of best-effort kernels is often longer than that of real-time kernels (e.g., VGG and DenseNet in Fig. 10), the throughput improvement of dynamic kernel padding may be trivial, even if the real-time tasks only use a few CUs.

6 Implementation

We first implemented and deployed REEF on AMD GPUs because of its open-source platform and ISA [26, 54], which can fully demonstrate the efficacy of reset-based preemption and dynamic kernel padding. REEF was implemented by extending Apache TVM [73] and AMD ROCm [3], with about 5,500 lines of C++ code. Beyond that, to further show the feasibility of REEF on closed-source GPUs, we also ported REEF-N, a restricted version of reset-based preemption, on NVIDIA GPUs with CUDA [52].

Model compiler. REEF extends Apache TVM [15], a machine learning compiler framework, with a code transformer, which mainly adds two modifications to the source code of DNN inference: (1) a *preemption flag*, which is injected into kernel arguments to lazily evict the kernel; (2) a set of *proxy kernels*, which is constructed for the padded kernels.

GPU runtime. For AMD GPUs, REEF builds the preemption module on HIP [63] of ROCm, a portable GPU runtime and programming library. similar to NVIDIA CUDA [52]. Specifically, REEF adds three new APIs to GPU runtime: (1) `hip_reset_hq`, which resets host queues and moves

commands to the GC thread; (2) `hip_set_stream_cap`, which limits the capacity of the device queue used by a GPU stream; (3) `hip_reset_kern`, which resets the compute units by using hardware mechanisms via the GPU driver in Linux [61].

For NVIDIA GPUs, REEF-N intercepts three CUDA APIs related to kernel launch and stream management, and adds the following operations: (1) `cuStreamCreate`, which creates a vHQ and links it to the created CUDA stream; (2) `cuKernelLaunch`, which buffers the launched kernel in the vHQ and transmits it to GPU runtime (i.e., CUDA [52]) in the background; (3) `cuStreamSynchronize`, which waits for GPU runtime to complete all launched kernel of the CUDA stream. Finally, REEF-N provides a new API `cuResetHQ` to reset vHQ by dequeuing all buffered kernels.

7 Evaluation

7.1 Experimental Setup

Testbed. The experiments were mainly conducted on a GPU server that consists of one Intel Core i7-10700 CPU (total 8 cores), 16 GB of DRAM, and one AMD Radeon Instinct MI50 GPU (60 CUs and 16GB of memory). The software environment of the server was configured with ROCm 4.3.0 [3], Apache TVM [73] 0.8.0, and Ubuntu 18.04. The hardware platform resembles the computational resources of autonomous vehicles [4, 71]. We further evaluate REEF-N on a closed-source GPU (NVIDIA V100 GPU) to demonstrate the generality of our approach, using the same server with CUDA 10.2 [52] installed.

Workloads. Inspired by YCSB [17, 18], we build a new DNN inference serving benchmark (DISB) that contains a suite of tools and five workloads: (A) low load, (B) high RT load, (C) high BE load, (D) multi-RT load, and (E) random load, summarized in Table 2. The real-time (RT) clients in DISB A–D uniformly send inference requests at a given frequency, which simulates real-time DNN applications in autonomous driving (e.g., obstacle recognition with cameras [7]), while the clients in DISB E send 20 requests per second with a Poisson arrival distribution, which simulates event-driven real-time DNN applications (e.g., speech recognition [32, 75]). Note that serving 220 RT requests per second sequentially for VGG model would saturate our testbed (see Fig. 1(d)). On the other hand, the closed-loop best-effort (BE) client continuously issues inference requests, which simulates a contention load on the GPU (e.g., driver monitoring).

Five representative DNN models are deployed in DISB, including ResNet-152 [30] (RNET), DenseNet-201 [35] (DNET), VGG-19 [68] (VGG), Inception v3 [69] (IN3), and DistilBert [66] (BERT), all generated by Apache TVM [15]. Each client always submits inference requests for a certain DNN model. Specifically, VGG is used by DISB A–C for their RT clients, and RNET is used by DISB A and B for their BE clients. Workloads with 5 RT/BE clients deploy all five DNN models in their clients separately, which simulates multiple DNN applications in a single scenario (e.g., autonomous vehicles [7, 41]).

Table 2: *DISB workload description.* #/model *denotes the number of clients and their DNN models.* [U/P] *denotes an arrival distribution (i.e., Uniform or Poisson).*

DISB	A	B	C	D	E
Num. of RT clients	1/VGG	1/VGG	1/VGG	5/ALL	5/ALL
Frequency (reqs/s)	100 [U]	220 [U]	100 [U]	20 [U]	20 [P]
Num. of BE clients	1/RNET	1/RNET	5/ALL	5/ALL	5/ALL

Furthermore, we use a real-world trace from an open autonomous driving platform (i.e., Apollo [7]) as the real-time workload, which provides a realistic arrival distribution of real-time tasks in autonomous driving. The trace was collected from the logs of the perception module [5] when running Apollo with SVL simulator [42, 65], and we selected the closest DNN models in terms of execution time from the above five models for the inference requests. Meanwhile, the same best-effort workload as DISB C–E is used, where five clients continuously issue different DNN inference requests.

Currently, each workload in DISB represents a particular mix of real-time and best-effort DNN inference tasks, the number of clients, and request frequency, which focuses on a particular point in the performance space. Users can further extend DISB with new workloads, or even some production traces from specific applications, to model more different scenarios.

Comparing targets. We compare REEF with typical scheduling approaches. **SEQ** sequentially runs each DNN inference task on the GPU with passive task preemption, which is adopted by Clockwork [28]. Specifically, when there are multiple tasks waiting in the queue, it prioritizes real-time tasks, but still needs to wait for the completion of launched best-effort tasks. **GPUStreams** runs both real-time and best-effort tasks simultaneously on the same GPU through multiple GPU streams, which is adopted by TensorRT [50]. As a reference, we further provide **RT-Only**, which represents the optimal end-to-end latency for real-time tasks, as it dedicates the GPU to real-time tasks.[4]

7.2 Overall Performance

We first compare the end-to-end latency of real-time tasks and the overall throughput of REEF with other approaches using DISB workloads and a real-world trace, as shown in Fig. 11.

Single BE Client (DISB A and B). For workloads with a single BE client, the performance impact of using SEQ or GPUStreams is relatively low, since GPU contention from best-effort tasks is not severe, either in terms of wait time (SEQ) or concurrent interference (GPUStreams). For DISB

[4] In this case, additional GPUs are dedicated to best-effort tasks, which also result in extra cost and energy consumption, as well as low GPU utilization.

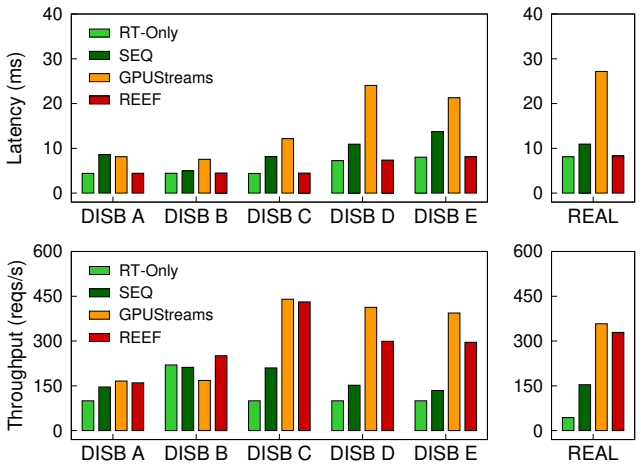


Fig. 11: *Comparison of (a) end-to-end real-time task latency, and (b) overall throughput (including both real-time and best-effort tasks) using different scheduling approaches.*

A, compared with RT-Only, SEQ and GPUStreams improve overall throughput by 1.46× and 1.66×, but also amplify real-time task latency by 1.95× and 1.84×, respectively. In contrast, REEF incurs negligible (0.5%) overhead on real-time task latency, but improves overall throughput by 1.60×, comparable to GPUStreams.

For DISB B, due to running real-time tasks more frequently, SEQ suffers 1.12× slowdown on real-time task latency, slightly better than DISB A, as it only has to wait for fewer best-effort tasks. However, its throughput only achieves 96% of RT-Only, since real-time tasks saturate the GPU and best-effort tasks have little chance to run. For similar reasons, the overall throughput of GPUStreams also drops to 76% of RT-Only, while its real-time task latency is still 1.70× higher than RT-only. Conversely, REEF can still limit the overhead on real-time task latency to 1% (about 60 µs) and provides a 1.14× speedup on overall throughput, thanks to our reset-based kernel preemption and dynamic kernel padding.

Multiple BE Clients (DISB C, D, and E). With the increase of best-effort workloads, the overall throughput of all approaches improve to varying degrees over RT-Only by sharing the GPU between two types of tasks. However, they have very different performance in terms of real-time task latency. Both SEQ and GPUStream make the same tradeoff between real-time task latency and overall throughput, differing only in the magnitude of the performance impact. For three workloads, SEQ improves overall throughput by 1.34× to 2.10×, but also amplifies real-time task latency by 1.51× to 1.86×. For GPUStreams, the above numbers become 3.94× to 8.19× and 2.65× to 3.31×.

Differently, REEF improves overall throughput as much as possible, based on the premise that real-time tasks should not be affected in any way. As a result, REEF offers almost the same real-time task latency as RT-Only in all workloads, with less than 1.5% overhead (0.1 ms). For overall throughput,

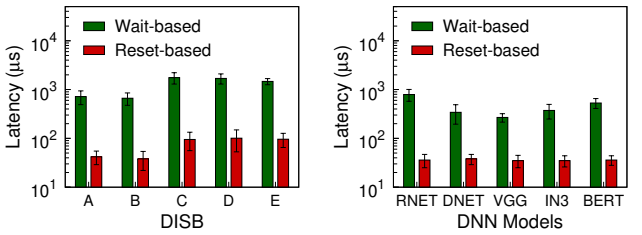


Fig. 12: *Comparison of preemption latency between reset-based and wait-based approaches (a) on DISB workloads, and (b) when preempting one DNN inference task of different DNN models.*

REEF provides a close result of GPUStreams on DISB C, since VGG is easy to be padded with most DNN models (see §5.2). On DISB D and E, the throughput of REEF is about 25% lower than that of GPUStreams, due to using a mix of five DNN models for real-time tasks, while DKP does not always work well on a few combinations of real-time and best-effort tasks (see §7.4 for details). However, REEF still outperforms RT-Only by 3.00× and 2.96×, respectively.

Real-world workload from Apollo (REAL). For the real-world workload, compared to RT-Only, SEQ and GPUStreams increase overall throughput by 3.6× and 8.3×, while amplifying the latency of real-time tasks by 1.35× and 3.35×, respectively. Due to the low load of real-time tasks in the real-world trace (about 43 reqs/s), REEF stays in normal mode to execute best-effort tasks concurrently most of the time, similar to GPUStreams. Therefore, compared to RT-Only, REEF achieves 7.7× throughput improvement with less than 2% latency overhead for real-time tasks, thanks to our reset-based preemption, which can preempt the GPU within tens of microseconds after the real-time task arrives.

7.3 DNN Inference Preemption

The vanilla wait-based preemption approach proposed in prior work [12] is not practical for DNN inference serving, since it only allows executing tasks one by one. Therefore, we extended it to allow concurrent inference serving by removing the limit on the amount of launched kernels and also implementing *lazy eviction*. This version is used as the baseline to demonstrate the efficiency of our reset-based preemption.

Preemption latency. Fig. 12(a) compares the preemption latency of two approaches. The reset-based preemption outperforms the wait-based approach by more than an order of magnitude for all DISB workloads, from 15.3× (DISB E) to 18.5× (DISB C). The main reason is that the wait-based approach has to passively wait for the completion of running kernels in CUs and the eviction of massive kernels in host and device queues, while the reset-based approach is able to proactively kill all kernels (usually much less) in these three places. As expected, both approaches take more time to handle multiple concurrent BE clients (DISB C, D, and E) than a singel BE client (DISB A and B).

Furthermore, we evaluate the preemption latency for di-

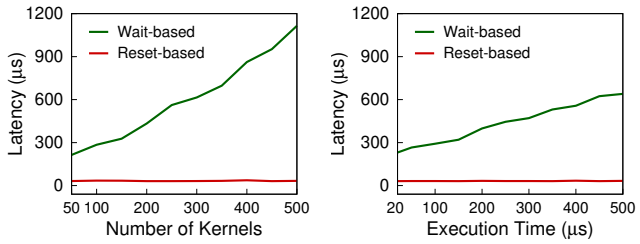


Fig. 13: *Comparison of preemption latency with the increase of (a) launched kernels and (b) kernel execution time.*

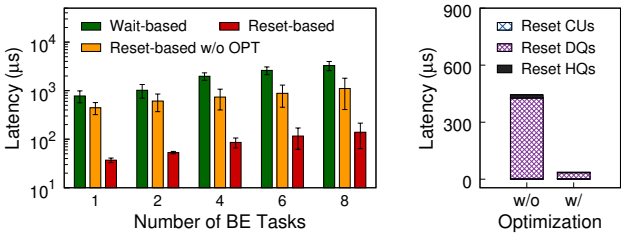


Fig. 14: *(a) Comparison of preemption latency with the increase of BE clients, and (b) the latency breakdown of the reset-based approach w/o and w/ optimizations.*

verse DNN models, where we use a single BE client to send inference requests for a given model and send a real-time request after a random time interval to preempt the GPU. As shown in Fig. 12(b), the wait-based preemption latency highly depends on the type of models, from 268 µs (VGG) to 790 µs (RNET), due to the difference in the number of kernels and the execution time (see Table 1). In contrast, the reset-based approach is not sensitive to DNN models and can preempt the GPU in the range of 35 µs to 38 µs for all five models.

To further investigate the impact of different model properties on the preemption latency, we simulate DNN models with different number of launched kernels and kernel execution times. By default, we set the number of launched kernels and the kernel execution time to 100 and 100 µs, respectively. As shown in Fig. 13, the preemption latency of wait-based approach raises linearly, while our reset-based preemption approach remains stable at very low latency (less than 40 µs). For wait-based approach, the preemption latency is significantly positively correlated with as number of launched kernels and the kernel execution time, since it has to wait for the eviction of launched kernels and the completion of running kernels. In contrast, the reset-based approach proactively resets the host and device queues in GPU runtime, as well as the CUs, where the cost is independent of model properties.

Optimizations. We propose two optimizations on the reset-based preemption approach, namely asynchronous memory reclamation and queue capacity restriction. To demonstrate the effect of optimizations, Fig. 14 shows the preemption latency with the increase of BE clients (RNET), and the latency breakdown for a single BE client. By enabling two optimizations, the preemption latency significantly drops by up to 92% (from 87%), as shown in Fig. 14(a). As a reference, even without optimization, the reset-based approach still outperforms wait-based approach by up to 3.0× (from 1.7×).

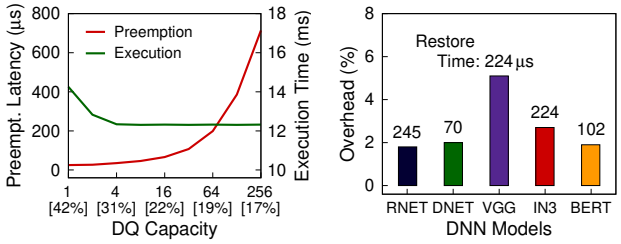


Fig. 15: *(a) The preemption latency and execution time with the increase of device queue capacity, where [%] shows the CPU utilization during normal execution, and (b) the restore overhead for different DNN models, where labels show the restore time (in µs).*

Since the two optimizations are used when resetting host and device queues, respectively, Fig. 14(b) breaks down the preemption latency to show the contribution of two optimizations separately. For a single BE client, using asynchronous memory reclamation reduces the latency of resetting host queue from 17 µs to 3 µs. Meanwhile, using queue capacity restriction further reduces the latency of resetting device queue from 424 µs to 31 µs. Note that using command processor to reset CUs is extremely fast (less than 3 µs).

Queue capacity. We restrict the device queue capacity to mitigate the overhead incurred by lazily evicting the remaining kernels in the queue (see §4.1 for details). However, reducing queue capacity also increases normal execution time and CPU utilization. Fig. 15(a) shows the preemption latency and normal execution time when serving RNET inferences as the queue capacity increases. When the device queue capacity increases from 1 to 4, the execution time reduces from 14.3 ms to 12.3 ms. However, when the capacity further increases, the change in execution time becomes trivial (less than 0.3%). Conversely, the preemption latency increases linearly with the queue capacity. Therefore, as a reasonable tradeoff between preemption latency and normal execution time, REEF adopts a default capacity of 4 for the device queue on our testbed, which has almost zero overhead for normal execution and provides acceptable preemption performance (about 30 µs). Finally, using a smaller device queue also results in higher CPU utilization. For instance, reducing the queue capacity from 256 to 4 increases CPU utilization from 17% to 31%.

Task restore. We further evaluate the execution time overhead of preempted tasks due to task restore. We use a single BE client to send inference requests; for each task, we randomly preempt and restore it. As shown in Fig. 15(b), the restore time for all DNN models is low, ranging from 70 µs to 245 µs, which mainly depends on the kernel execution time of DNN models (see Fig. 10). Note that REEF redundantly executes at most five kernels for restoring preempted tasks, thanks to the queue capacity restriction. Further, the execution time overhead is about 2% for all DNN models, except for VGG (5.1%), as it has the fewest kernels (55), and its kernel

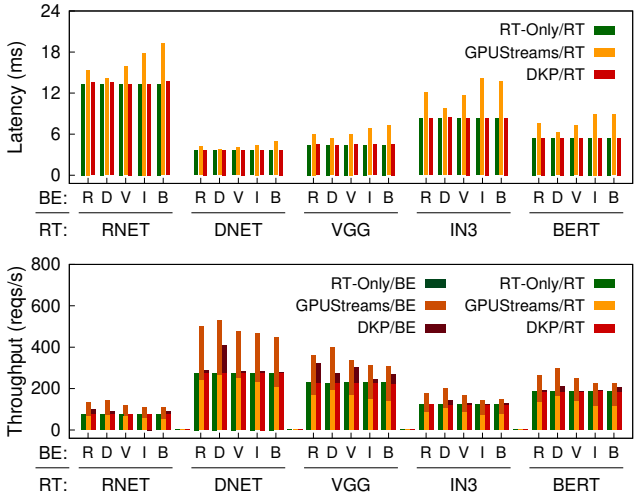


Fig. 16: *Comparison of (a) end-to-end latency of RT tasks and (b) overall throughput using different concurrent execution schemes.*

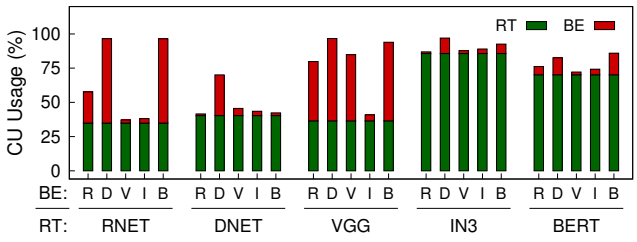


Fig. 17: *The average CU usage for running real-time and best-effort kernels with different combinations of DNN models using dynamic kernel padding.*

execution time is longer (see Fig. 10).

7.4 Dynamic Kernel Padding

To study the efficacy of dynamic kernel padding, we use a high contention workload, where one RT client and one BE client simultaneously send requests at a high-enough frequency to keep the GPU busy. RT-Only serves only real-time tasks to ensure optimal (real-time) task latency, while GPUStreams serves both types of requests concurrently to achieve the highest overall throughput. Differently, dynamic kernel padding also serves only real-time tasks but pads best-effort tasks to avoid starvation and improve overall throughput.

Performance. Fig. 16 reports the experimental results for one-to-one combinations among five DNN models using above workload. As expected, GPUStreams significantly amplifies real-time task latency by an average of 1.35×, ranging from 1.04× to 1.70×, due to severe interference from concurrent best-effort tasks. However, REEF is able to provide almost optimal latency to real-time tasks, with an average overhead of just 1% (up to 3%).

For overall throughput, we separately report the throughput of real-time tasks and the normalized throughput of best-effort

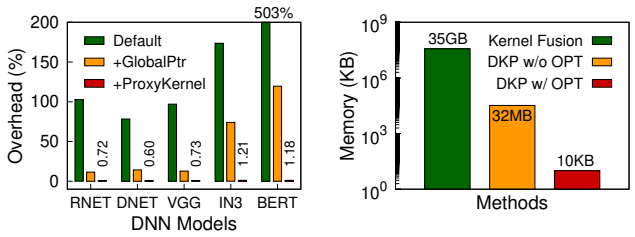


Fig. 18: *Comparison of (a) execution time overhead and (b) memory overhead for padded kernels using different optimizations.*

tasks.[5] For RT-Only, the GPU is busy serving real-time tasks, so the throughput of best-effort tasks is zero (even if RT-Only is willing to serve them). Although GPUStreams increases overall throughput by an average of 1.52×, the throughput of real-time tasks drops by 24.4% on average, due to severe interference in concurrent execution. Conversely, REEF first guarantees throughput for real-time tasks and then leverages dynamic kernel padding to increase overall throughput. The performance improvement mainly depends on two conditions. First, the execution of real-time tasks on the GPU leaves room for improvement. As shown in Fig. 17, the real-time kernels in IN3 and BERT use an average of 85% and 70% of CUs, respectively. Therefore, dynamic kernel padding hardly improves such cases, increasing just 6% on average. Note that GPUStreams can still improve overall throughput of them, but also greatly sacrifices the performance of real-time tasks. Second, the execution time of best-effort kernels must be shorter than that of the padded real-time kernels. This explains why REEF can achieve large improvement (1.41×) by padding VGG with RNET, but not vice versa, which is also confirmed by the increase of CU usage (BE) in Fig. 17

Optimizations. To investigate the impact of optimizations on both performance and memory usage, we first evaluate the overhead using different implementations of the function pointer on the GPU. We measured such overhead by launching real-time kernels through the dkp kernel without padding any best-effort kernels. As shown in Fig. 18(a), the default function pointer implementation (Default) incurs execution time overhead from 78% up to 503% for real-time tasks with different DNN models. By using the global function pointer (GlobalPtr), the overhead is significantly reduced to 46.4% on average (from 11.5% to 120%), as it eliminates the limit on the number of registers for device function pointers and avoids additional register saving and restoring during the function call. Finally, the overhead drops to 0.8% on average (1.21% at most) by using proxy kernel (ProxyKernel), which can dynamically allocate registers to each kernel and maximize CU occupancy. The minimal overhead comes from the logic branch of CU partition and the initial state preparation for global function pointers.

We further evaluate the impact of optimizations on reduc-

[5]The throughput of best-effort tasks is normalized to that of real-time tasks, following the formula: $throughput_{\text{BE}} \times (latency_{\text{BE}} \,/\, latency_{\text{RT}})$.

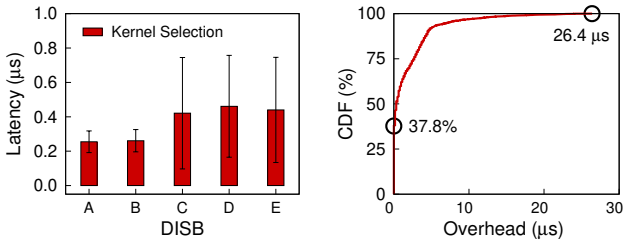


Fig. 19: *(a) The execution time of kernel selection for DISB A-E and (b) the CDF of execution time overhead for real-time kernels using dynamic kernel padding.*

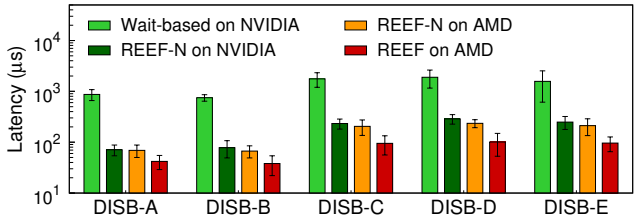


Fig. 20: *Comparison of preemption latency on NVIDIA and AMD GPUs using different preemption schemes with DISB workloads.*

ing GPU memory usage. As shown in Fig. 18(b), using static kernel fusion (Kernel Fusion) requires over 35 GB of GPU memory to store the fused kernels for five DNN models—all combinations of no more than three kernels, which even exceeds the memory capacity of most commodity GPUs. REEF proposes proxy kernels (DKP w/o OPT) to reduce GPU memory usage to about 32 MB. Finally, generating proxy kernels to cover all possible CU occupancies (DKP w/ OPT), instead of all possible register configurations, can dramatically reduce GPU memory usage to only 10 KB.

Kernel selection. Fig. 19(a) shows the average time of kernel selection for DISB A-E during dynamic kernel padding. For workloads with a single BE client (DISB A and B), REEF takes about 0.2 µs to select best-effort kernels for the given real-time kernel. The selection time increases to 0.4 µs for workloads with multiple BE clients (DISB C, D, and E) due to more candidates. In general, the cost of kernel selection is quite trivial and can be easily hidden by kernel execution.

To further study the accuracy of kernel selection, we evaluate the execution time overhead for the real-time kernel due to padding best-effort kernels on all DISB workloads. As shown in Fig. 19(b), over 37% of real-time kernels are not negatively impacted by concurrent execution with best-effort kernels, and the overhead of more than 90% real-time kernels is still less than 4 µs. The increase of execution time is mainly due to the contention on GPU memory and shared L2 cache.

7.5 Closed-source GPUs

Finally, we evaluate REEF-N, a restricted version of reset-based preemption using DISB workloads on both NVIDIA and AMD GPUs, and compare it to the wait-based approach and REEF, respectively. As shown in Fig. 20, even if REEF-N does not reset CUs to proactively kill running kernels, the preemption latency just ranges from 71µs to 288µs, which still outperforms the wait-based approach by up to 12.3× (from 6.3×) on the NVIDIA GPU. By comparing REEF-N and REEF on the AMD GPU, we observe that killing running kernels proactively further contributes to an average speedup of 2.0× in preemption latency, especially for preempting concurrent tasks (e.g., 2.3× for DISB C). In addition, the performance of REEF-N is close on two GPUs.

8 Discussion

Assumption of idempotence. The reset-based preemption in REEF is based on the assumption that each kernel in DNN inference should be idempotent. Currently, all DNN inference kernels we encountered, a total of 320 kernels from 11 models [72], are shown to be idempotent. However, readers might be interested in whether our approach still works with kernels without the idempotence assumption. Strictly speaking, the reset-based preemption demands that the kernel always produces the same output for the same input no matter it has been retried or not. Therefore, a transactionization approach [40] can be used to transform non-idempotent kernels into idempotent ones if necessary. Furthermore, since only best-effort kernels may be preempted in REEF, this transformation only sacrifices the performance of transformed kernels (i.e., best-effort kernels) to ensure that real-time kernels can be instantly executed upon arrival with no performance penalty. We leave the incorporation of this technique to future work until we actually encounter non-idempotent DNN kernels.

Restrictions on kernel selection. The current kernel selection policy is effective but conservative, since the primary goal of REEF is to avoid performance interference with real-time tasks. An obvious limitation is the constraint that the execution time of best-effort kernels must be shorter than that of the padded real-time kernel, which limits room for improvement in overall throughput. We found that the GPU kernel can be tailored towards shorter execution time per block by using more thread blocks during model compilation. For example, Apache TVM automatically tunes the number of thread blocks for overall performance, but also allows developers to customize it [38]. Currently, the overall throughput improvement of REEF is largely attributed to enabling instant kernel preemption, which allows the idle GPU to perform best-effort tasks. Thus, we leave it to future work to overcome the restriction on kernel selection. Furthermore, the policy does not consider the contention for GPU memory between real-time and best-effort kernels, since it is still sufficient for running multiple DNN inference tasks. We also leave it to future work.

Future GPU APIs and runtime. We leverage several subtle hacks on the GPU runtime to enable µs-scale reset-based preemption on commodity GPUs. Our work also informs the design of future GPU APIs and runtime. First, given that com-

modity GPUs are generally capable of resetting compute units (CUs), a separate GPU API to precisely reset CUs is feasible and would be useful to kill and restore all running kernels. Second, we propose a new GPU API that instructs the command processor to discard fetched kernels and stop fetching more kernels from the device queue (DQs). Based on it, DQs can be proactively reset with a hardware-software co-design, replacing our software-only solution (i.e., lazy eviction). Finally, the GPU runtime could provide a high-level API for developers to reset the GPU stream, by discarding kernels buffered in internal data structures (e.g., host queues) and resetting the GPU via two new APIs. We believe that these extensions can greatly simplify implementation, even fully implementing reset-based preemption on closed-source GPUs, and further improve performance, for example instantly preempting the GPU in 10 µs.

9 Related Work

DNN inference serving systems. Prior model serving systems [21, 25, 29, 53, 79] mainly focus on meeting service-level objectives (SLO), typically in the tens of milliseconds [22, 31, 81], and improving overall throughput of datacenter applications. Clockwork [28] leverages the latency predictability of DNN inference to achieve low tail latency. It runs inferences sequentially on dedicated GPUs to provide predictable performance. Clipper [20] and Nexus [67] enables batching inferences on the same model to improve GPU utilization and inference throughput. Abacus [22] enables simultaneous DNN inferences by accurately predicting the latency of the overlapped operators. INFaaS [64] can automatically select the right variant with different optimizations for each inference to meet diverse SLOs. However, the latency SLOs for datacenter applications are much more relaxed than those for real-time systems, for example 2× of their solo-run latencies [22]. Therefore, using non-preemptive scheduling or batching scheme is effective for datacenter applications, but not for real-time scenarios (e.g., autonomous vehicles). Furthermore, the design of REEF is orthogonal to the above distributed serving systems. Two key mechanisms in REEF can also be integrated into them to improve per-GPU throughput and preserve low latency for real-time inferences.

GPU kernel preemption. Apart from the software preemption techniques, prior work also has proposed hardware enhancement to support preemptive GPU scheduling [44, 56, 70]. An intuitive solution is to support context switch on GPUs [70]. However, it is far more expensive on GPU than CPU due to the large context (e.g., a large amount of registers). Zhen et al. [44] proposed lightweight context switching to avoid unnecessary register saving. Tanasić et al. [70] extended the hardware to passively preempt a streaming multiprocessor (SM) of GPU by stopping issuing new thread blocks. Chimera [56] further proposed SM flushing to instantly preempt an SM when detecting idempotent execution. Differently, our approach retrofits existing hardware mechanism and requires no modification on the GPU to implement instant preemption.

GPU multitasking. There have been many efforts to concurrently execute multiple GPU kernels for high throughput [27, 43, 55, 57, 74, 76]. For DNN computation, Rammer [47] takes a holistic approach to exploit both inter- and intra-kernel parallelisms at compile time, which uses static kernel fusion [74] to enforce the CU assignments of the concurrent kernels. However, static kernel fusion requires the fused kernels to be known at compile time, which is not applicable for dynamic task scheduling in REEF. REEF proposes dynamic kernel padding to allow making scheduling decisions at runtime. Prior work has also proposed approaches to model and predict the slowdown of concurrent kernel execution [13, 14, 86, 88]. DASE [34] models the memory contention of concurrent kernels. Themis [87] uses a neural network to predict the performance interference. The prediction can help make scheduling decisions to match the latency requirements of real-time kernels. However, the prediction cannot always be accurate, and the slowdown actually happens. Differently, dynamic kernel padding in REEF enforces concurrent kernels to use only GPU resources leftover from the real-time kernel. Currently, REEF mainly focuses on GPU computational resources (i.e., CUs) and assumes that other resources are sufficient (e.g., GPU memory and bandwidth). We leave it as future work.

10 Conclusion

This paper presented REEF, the first DNN inference serving system for commodity GPUs. It enables microsecond-scale kernel preemption and controlled concurrent execution in GPU scheduling to achieve real time and work conserving. First, REEF can launch a real-time kernel on the GPU by proactively killing and restoring best-effort kernels at microsecond-scale. Second, REEF can dynamically pad the real-time kernel with appropriate best-effort kernels to fully exploit the GPU with negligible overhead. In addition, we built a new benchmark (DISB) for DNN inference serving that contains diverse workloads and a real-world trace. Evaluation using DISB and microbenchmarks confirmed the efficacy and efficiency of REEF on AMD and NVIDIA GPUs.

11 Acknowledgment

We sincerely thank our shepherd Dejan Kostić and the anonymous reviewers for their insightful comments and feedback, and Xiaoniu Song for sharing his experience in preparing the Artifact Evaluation. This work was supported in part by the National Natural Science Foundation of China (No. 61925206, 62132014), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and Shanghai AI Laboratory. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] Jacob Adriaens, Katherine Compton, Nam Sung Kim, and M. Schulte. The Case for GPGPU Spatial Multitasking. *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012.

[2] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions. *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 267–280, 2020.

[3] AMD ROCm. AMD ROCm Platform Documentation. https://rocmdocs.amd.com/, 2022.

[4] Apollo Auto. Apollo: Architecture/Hardware Connection. https://github.com/ApolloAuto/apollo, 2022.

[5] Apollo Auto. Apollo Perception Module. https://github.com/ApolloAuto/apollo/tree/master/modules/perception, 2022.

[6] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 499–514, November 2020.

[7] Baidu. Apollo. https://apollo.auto/, 2022.

[8] C. Basaran and K. Kang. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *24th Euromicro Conference on Real-Time Systems*, ECRTS'12, pages 287–296, 2012.

[9] Karsten Behrendt, Libor Novak, and Rami Botros. A Deep Learning Approach to Traffic Lights: Detection, Tracking, and Classification. *IEEE International Conference on Robotics and Automation*, pages 1370–1377, 2017.

[10] N. Capodieci, R. Cavicchioli, M. Bertogna, and Aingara Paramakuru. Deadline-Based Scheduling for GPU with Preemption Support. *IEEE Real-Time Systems Symposium*, pages 119–130, 2018.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. *IEEE International Symposium on Workload Characterization*, pages 44–54, 2009.

[12] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU. *22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017.

[13] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. *Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[14] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. *Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[15] T. Chen, T. Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, L. Ceze, Carlos Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'18, 2018.

[16] Green Car Congress. New ultrafast camera for self-driving vehicles and drones. https://www.greencarcongress.com/2017/02/20170217-ntu.html, 2017.

[17] Brian F. Cooper. YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, 2022.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143—-154, 2010.

[19] Alexander Craik, Yongtian He, and José Luis Contreras-Vidal. Deep Learning for Electroencephalogram (EEG) Classification Tasks: A Review. *Journal of neural engineering*, 16(3), 2019.

[20] D. Crankshaw, Xin Wang, Giulio Zhou, M. Franklin, Joseph E. Gonzalez, and I. Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, 2017.

[21] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Ming Guo. Ebird: Elastic Batch for Improving Responsiveness and Throughput of Deep Learning Services. *IEEE 37th International Conference on Computer Design*, pages 497–505, 2019.

[22] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Enable Simultaneous DNN Services Based on Deterministic Operator Overlap and Precise Latency Prediction. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[23] ROCm documentation. GCN Native ISA LLVM Code Generator: Kernel Dispatch. https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Native-ISA.html, 2022.

[24] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A Guide to Deep Learning in Healthcare. *Nature medicine*, 25(1):24–29, 2019.

[25] Jiarui Fang, Yang Yu, Chen liang Zhao, and Jie Zhou. TurboTransformers: An Efficient GPU Serving System for Transformer Models. *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[26] AMD GPUOpen. AMD GPU ISA documentation. https://gpuopen.com/documentation/amd-isa-documentation, 2021.

[27] Chris Gregg, Jonathan Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *4th USENIX Workshop on Hot Topics in Parallelism*, HotPar'12, 2012.

[28] A. Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, 2020.

[29] Johann Hauswald, Yiping Kang, Michael Laurenzano, Quan Chen, Cheng Li, Trevor N. Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. DjiNN and Tonic: DNN as A Service and Its Implications for Future Warehouse Scale Computers. *ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, pages 27–40, 2015.

[30] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[31] Jeremy Hermann and Mike Del Balso. Meet Michelangelo: Uber's Machine Learning Platform. https://eng.uber.com/michelangelo-machine-learning-platform/, 2017.

[32] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdelrahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[33] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. GRNN: Low-Latency and Scalable RNN Inference on GPUs. *14th European Conference on Computer Systems*, 2019.

[34] Qingda Hu, J. Shu, Jie Fan, and Youyou Lu. Run-Time Performance Estimation and Fairness-Oriented Scheduling Policy for Concurrent GPGPU Applications. *45th International Conference on Parallel Processing*, pages 57–66, 2016.

[35] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2261–2269, 2017.

[36] Saksham Jain, Iljoo Baek, Shige Wang, and R. Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 29–41, 2019.

[37] Won-Seok Jang, Hansaem Jeong, Kyungtae Kang, Nikil D. Dutt, and Jong-Chan Kim. R-TOD: Real-Time Object Detector with Minimized End-to-End Delay for Autonomous Driving. *IEEE Real-Time Systems Symposium*, pages 191–204, 2020.

[38] Ziheng Jiang. Schedule Primitives in TVM. https://tvm.apache.org/docs/how_to/work_with_schedules/schedule_primitives.html.

[39] Hyeonsu Lee, Hyunjune Kim, Cheolgi Kim, Hwansoo Han, and Euiseong Seo. Idempotence-Based Preemptive GPU Kernel Scheduling for Embedded Systems. *IEEE Transactions on Computers*, 70:332–346, 2021.

[40] Hyeonsu Lee, Jaehun Roh, and Euiseong Seo. A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS'18, pages 202–213, 2018.

[41] TIMOTHY B. LEE. Tesla's autonomy event:Impressive progress with an unrealistic timeline. https://arstechnica.com/cars/2019/04/teslas-autonomy-event-impressive-progress-with-an-unrealistic-timeline/, 2019.

[42] LG Electronics Inc. Running Apollo 5.0 with SVL Simulator. https://www.svlsimulator.com/docs/system-under-test/apollo5-0-instructions/, 2022.

[43] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, R. Goh, and Deming Chen. Efficient GPU Spatial-Temporal Multitasking. *IEEE Transactions on Parallel and Distributed Systems*, 26:748–760, 2015.

[44] Zhen Lin, L. Nyland, and Huiyang Zhou. Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching. *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 898–908, 2016.

[45] LLVM. User Guide for AMDGPU Backend. https://llvm.org/docs/AMDGPUUsage.html, 2021.

[46] Justin Luitjens. CUDA Streams—Best Practices and Common Pitfalls. http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

[47] Lingxiao Ma, Z. Xie, Zhi Yang, J. Xue, Youshan Miao, Wei Cui, W. Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 881–897, 2020.

[48] Pavlo Molchanov, Shalini Gupta, Kihwan Kim, and Kari Pulli. Multi-sensor System for Driver's Hand-gesture Recognition. *11th IEEE International Conference and Workshops on Automatic Face and Gesture Recognition*, 1:1–8, 2015.

[49] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating Deep Learning Workloads Through Efficient Multi-model Execution. In *NeurIPS Workshop on Systems for Machine Learning*, page 20, 2018.

[50] NVIDIA. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[51] NVIDIA. NVIDIA Tesla P100. http://www.nvidia.com/object/pascal-architecture-whitepaper.html, 2016.

[52] NVIDIA. CUDA Toolkit: Develop, Optimize and Deploy GPU-Accelerated Apps. https://developer.nvidia.com/cuda-toolkit, 2021.

[53] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[54] Nathan Otterness and James H. Anderson. AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads. In *32nd Euromicro Conference on Real-Time Systems*, ECRTS'20, 2020.

[55] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, 2013.

[56] J. Park, Yongjun Park, and S. Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. *Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[57] J. Park, Yongjun Park, and S. Mahlke. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. *Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[58] Reid Pinkham, Andrew Berkovich, and Zhengya Zhang. Near-Sensor Distributed DNN Processing for Augmented and Virtual Reality. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2021.

[59] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[60] Steve Rennich. CUDA C/C++ Streams and Concurrency. https://developer.download.nvidia.cn/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[61] ROCm Core Technology. AMD GPU kernel driver with KFD. https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver, 2022.

[62] ROCm Core Technology. AMD GPU kernel driver with KFD: unmap_queues_cpsch. https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver/blob/master/drivers/gpu/drm/amd/amdkfd/kfd_device_queue_manager.c, 2022.

[63] ROCm Developer Tools. Hip: C++ heterogeneous-compute interface for portability. https://github.com/ROCm-Developer-Tools/HIP, 2022.

[64] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated Model-less Inference Serving. In *USENIX Annual Technical Conference*, ATC'21, pages 397–411, 2021.

[65] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, Eugene Agafonov, Tae Hyung Kim, Eric Sterner, Keunhae Ushiroda, Michael Reyes, Dmitry Zelenkovsky, and Seonman Kim. LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving. In *IEEE 23rd International Conference on Intelligent Transportation Systems Conference*, ITSC'20, pages 1–6, 2020.

[66] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *CoRR*, abs/1910.01108, 2019.

[67] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis. *27th ACM Symposium on Operating Systems Principles*, 2019.

[68] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[69] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[70] I. Tanasić, Isaac Gelado, Javier Cabezas, A. Ramírez, N. Navarro, and M. Valero. Enabling Preemptive Multiprogramming on GPUs. *ACM/IEEE 41st International Symposium on Computer Architecture*, pages 193–204, 2014.

[71] TESLARATI. AMD confirms Tesla's new Model S and Model X will boast RDNA 2 GPUs. https://www.teslarati.com/tesla-model-s-model-x-mcu3-specs-amd-gpu-confirmed-video/, 2021.

[72] Apache TVM. A test suite of DNN models. https://github.com/apache/tvm/tree/v0.8/python/tvm/relay/testing, 2021.

[73] Apache TVM. Apache TVM: An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators. https://tvm.apache.org/, 2021.

[74] Guibin Wang, Yisong Lin, and Wei Yi. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. *IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350, 2010.

[75] Yuxuan Wang, RJ Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, et al. Tacotron: A Fully End-to-end Text-to-speech Synthesis Model. *arXiv preprint arXiv:1703.10135*, 2017.

[76] Zhenning Wang, J. Yang, R. Melhem, B. Childers, Youtao Zhang, and M. Guo. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-grained Sharing. *IEEE International Symposium on High Performance Computer Architecture*, pages 358–369, 2016.

[77] Bo Wu, Xu Liu, Xiaobo Zhou, and C. Jiang. Flep: Enabling flexible and efficient preemption on gpus. *Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[78] Yecheng Xiang and Hyoseung Kim. Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference. *IEEE Real-Time Systems Symposium*, pages 392–405, 2019.

[79] Feng Yan, Yuxiong He, Olatunji Ruwase, and Evgenia Smirni. Efficient Deep Neural Network Serving: Fast and Furious. *IEEE Transactions on Network and Service Management*, 15:112–126, 2018.

[80] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F. Donelson Smith, James H. Anderson, and Jan-Michael Frahm. Re-Thinking CNN Frameworks for Time-Sensitive Autonomous-Driving Applications: Addressing an Industrial Challenge. *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 305–317, 2019.

[81] Wai Chee Yau. How Zendesk Serves TensorFlow Models in Production. https://zendesk.engineering/how-zendesk-serves-tensorflow-models-in-production-751ee22f0f4b, 2017.

[82] T. Yeh, Matthew D. Sinclair, Bradford M. Beckmann, and Timothy G. Rogers. Deadline-Aware Offloading for High-Throughput Accelerators. *IEEE International Symposium on High-Performance Computer Architecture*, pages 479–492, 2021.

[83] Juheon Yi and Youngki Lee. Heimdall: mobile gpu coordination platform for augmented reality applications. In *26th Annual International Conference on Mobile Computing and Networking*, MobiCom'20, pages 1–14, 2020.

[84] Sebastian Zepf, Javier Hernandez, Alexander Schmitt, Wolfgang Minker, and Rosalind W. Picard. Driver Emotion Recognition for Intelligent Vehicles. *ACM Computing Surveys*, 53:1–30, 2020.

[85] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Erran L. Li, Tiancheng Lou, and Jishen Zhao. Towards Safety-Aware Computing System Design in Autonomous Vehicles. *ArXiv*, abs/1905.08453, 2019.

[86] Wenyi Zhao, Quan Chen, and M. Guo. KSM: Online Application-Level Performance Slowdown Prediction for Spatial Multitasking GPGPU. *IEEE Computer Architecture Letters*, 17:187–191, 2018.

[87] Wenyi Zhao, Quan Chen, H. Lin, Jianfeng Zhang, Jingwen Leng, C. Li, Wenli Zheng, Linlin Li, and M. Guo. Themis: Predicting and Reining in Application-Level Slowdown on Spatial Multitasking GPUs. *IEEE International Parallel and Distributed Processing Symposium*, pages 653–663, 2019.

[88] Xia Zhao, Magnus Jahre, and L. Eeckhout. HSM: A Hybrid Slowdown Model for Multitasking GPUs. *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[89] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor : Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, 2020.

[90] H. Zhou, G. Tong, and Cong Liu. GPES: A Preemptive Execution System for GPGPU Computing. *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97, 2015.

[91] Husheng Zhou, Soroush Bateni, and Cong Liu. S3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. *2018 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 190–201, 2018.

A Artifact Appendix

This artifact provides the source code of REEF, a detailed readme, and scripts to reproduce the main experimental results from the OSDI 2022 paper—"Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences" by M. Han, H. Zhang, R. Chen, and H. Chen. REEF is the first GPU-accelerated DNN inference serving system that enables microsecond-scale kernel preemption and controlled concurrent execution in GPU scheduling. We provide instructions to build the software package and run experiments. Our artifact obtained the "Artifacts Available", "Artifacts Functional" and "Artifacts Reproduced" badges from the Artifact Evaluation process of OSDI 2022. The DOI of our artifact is https://doi.org/10.5281/zenodo.6586106.

Artifact repository. All project source code, including full instructions on how to build and run the main experiments on REEF and benchmarks is available in the following git repository: https://github.com/SJTU-IPADS/reef-artifacts/tree/osdi22-ae.

Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning

Lianmin Zheng[1,*] Zhuohan Li[1,*] Hao Zhang[1,*] Yonghao Zhuang[4]
Zhifeng Chen[3] Yanping Huang[3] Yida Wang[2] Yuanzhong Xu[3] Danyang Zhuo[6] Eric P. Xing[5]
Joseph E. Gonzalez[1] Ion Stoica[1]

[1]*UC Berkeley* [2]*Amazon Web Services* [3]*Google* [4]*Shanghai Jiao Tong University*
[5]*MBZUAI, Carnegie Mellon University* [6]*Duke University*

Abstract

Alpa automates model-parallel training of large deep learning (DL) models by generating execution plans that unify data, operator, and pipeline parallelism. Existing model-parallel training systems either require users to manually create a parallelization plan or automatically generate one from a limited space of model parallelism configurations. They do not suffice to scale out complex DL models on distributed compute devices. Alpa distributes the training of large DL models by viewing parallelisms as two hierarchical levels: inter-operator and intra-operator parallelisms. Based on it, Alpa constructs a new hierarchical space for massive model-parallel execution plans. Alpa designs a number of compilation passes to automatically derive efficient parallel execution plans at each parallelism level. Alpa implements an efficient runtime to orchestrate the two-level parallel execution on distributed compute devices. Our evaluation shows Alpa generates parallelization plans that match or outperform hand-tuned model-parallel training systems even on models they are designed for. Unlike specialized systems, Alpa also generalizes to models with heterogeneous architectures and models without manually-designed plans. Alpa's source code is publicly available at https://github.com/alpa-projects/alpa.

1 Introduction

Several of the recent advances [10, 22, 49] in deep learning (DL) have been a direct result of significant increases in model size. For example, scaling language models, such as GPT-3, to hundreds of billions of parameters [10] and training on much larger datasets enabled fundamentally new capabilities.

However, training these extremely large models on distributed clusters currently requires a significant amount of engineering effort that is specific to both the model definition and the cluster environment. For example, training a large transformer-based language model requires heavy tuning and careful selection of multiple parallelism dimensions [40]. Training the large Mixture-of-Expert (MoE) transformers model [16, 31] on TPU clusters requires manually tuning the partitioning axis for each layer, whereas training the same model on an AWS GPU cluster calls for new pipeline schemes that can depend on the choices of partitioning (§8.1).

More generally, efficient large-scale model training requires tuning a complex combination of data, operator, and pipeline parallelization approaches at the granularity of the individual tensor operators. Correctly tuning the parallelization strategy has been shown [30, 33] to deliver an order of magnitude improvements in training performance, but depends on strong machine learning (ML) and system expertise.

Automating the parallelization of large-scale models would significantly accelerate ML research and production by enabling model developers to quickly explore new model designs without regard for the underlying system challenges. Unfortunately, it requires navigating a complex space of plans that grows exponentially with the dimensions of parallelism and the size of the model and cluster. For example, when all parallelism techniques are enabled, figuring out the execution plan involves answering a web of interdependent questions, such as how many data-parallel replicas to create, which axis to partition each operator along, how to split the model into pipeline stages, and how to map devices to the resulting parallel executables. The interplay of different parallelization methods and their strong dependence on model and cluster setups form a combinatorial space of plans to optimize. Recent efforts [17, 38, 55] to automatically parallelize model training are constrained to the space of a single model-parallelism approach, or rely on strong assumptions on the model and cluster specifications (§2.1).

Our key observation is that we can organize different parallelization techniques into a *hierarchical space* and map these parallelization techniques to the *hierarchical structure* of the compute cluster. Different parallelization techniques have different bandwidth requirements for communication, while a typical compute cluster has a corresponding structure: closely located devices can communicate with high bandwidth while distant devices have limited communication bandwidth.

*Lianmin, Zhuohan, and Hao contributed equally. Part of the work was done when Lianmin interned at Amazon and Zhuohan interned at Google.

With this observation in mind, in this paper, we take a different view from conventional data and model parallelisms, and re-categorize ML parallelization approaches as *intra-operator* and *inter-operator* parallelisms. Intra-operator parallelism partitions ML operators along one or more tensor axes (batch or non-batch) and dispatches the partitions to distributed devices (Fig. 1c); inter-operator parallelism, on the other hand, slices the model into disjoint stages and pipelines the execution of stages on different sets of devices (Fig. 1d). They take place at two different granularities of the model computation, differentiated by whether to partition operators.

Given that, a parallel execution plan can be expressed *hierarchically* by specifying the plan in each parallelism category, leading to a number of advantages. First, intra- and inter-operator parallelisms feature distinct characteristics: intra-operator parallelism has better device utilization, but results in communicating at every split and merge of partitioned operators, per training iteration; whereas inter-operator parallelism only communicates between adjacent stages, which can be light if sliced properly, but incurs device idle time due to scheduling constraints. We can harness the asymmetric nature of communication bandwidth in a compute cluster, and map intra-operator parallelism to devices connected with high communication bandwidth, while orchestrating the inter-operator parallelism between distant devices with relatively lower bandwidth in between. Second, this hierarchical design allows us to solve each level near-optimally as an individual tractable sub-problem. While the joint execution plan is not guaranteed globally optimal, they demonstrate strong performance empirically for training various large models.

Guided by this new problem formulation, we design and implement Alpa, the first compiler that automatically generates parallel execution plans covering all data, operator, and pipeline parallelisms. Given the model description and a cluster configuration, Alpa achieves this by partitioning the cluster into a number of *device meshes*, each of which contains devices with preferably high-bandwidth connections, and partitioning the computation graph of the model into *stages*. It assigns stages to device meshes, and automatically orchestrates intra-operator parallelisms on a device mesh and inter-operator parallelisms between device meshes.

In summary, we make the following contributions:

• We construct a two-level parallel execution plan space (Fig. 1e) where plans are specified hierarchically using inter- and intra-operator parallelisms.

• We design tractable optimization algorithms to derive near-optimal execution plans at each level.

• We implement Alpa, a compiler system for distributed DL on GPU clusters. Alpa features: (1) a set of compilation passes that generate execution plans using the hierarchical optimization algorithms, (2) a new runtime architecture that orchestrates the inter-op parallelism between device meshes, and (3) a number of system optimizations that improve compilation and address cross-mesh communication.

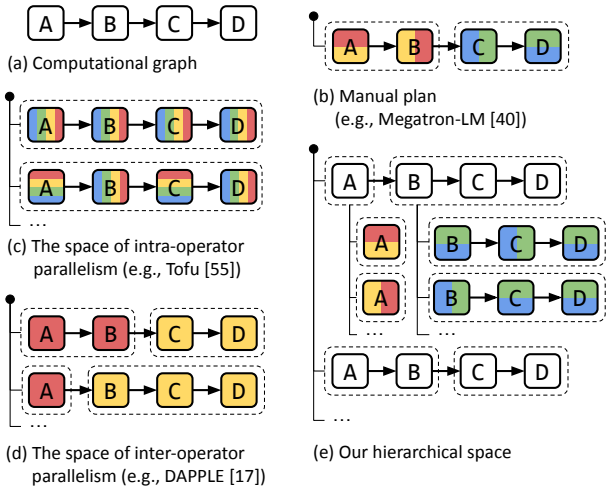


Figure 1: Generation of parallelization plans for a computational graph shown in (a). Different colors represent different devices, dashed boxes represent pipeline stages. (b) creates the plan manually. (c) and (d) automatically generate plans using only one of intra- and inter-operator parallelisms. (e) shows our approach that creates a hierarchical space to combine intra- and inter-operator parallelisms.

• We evaluate Alpa on training large models with billions of parameters. We compare Alpa with state-of-the-art distributed training systems on an Amazon EC2 cluster of 8 p3.16xlarge instances with 64 GPUs. On GPT [10] models, Alpa can match the specialized system Megatron-LM [40,49]. On GShard MoE models [31], compared to a hand-tuned system Deepspeed [45], Alpa achieves a 3.5× speedup on 2 nodes and a 9.7× speedup on 4 nodes. Unlike specialized systems, Alpa also generalizes to models without manual strategies and achieves an 80% linear scaling efficiency on Wide-ResNet [59] with 4 nodes. This means developers can get efficient model-parallel execution of large DL models out-of-the-box using Alpa.

2 Background: Distributed Deep Learning

DL computation is commonly represented by popular ML frameworks [1,9,42] as a dataflow graph. Edges in the graph represent multi-dimensional tensors; nodes are computational operators, such as matrix multiplication (matmul), that transform input tensors into output tensors. Training a DL model for one iteration consists of computing a loss by *forwarding* a batch of data through the graph, deriving the updates via a reverse *backward* pass, and applying the updates to the parameters via *weight update* operations. In practice, model developers define the dataflow graph. An execution engine then optimizes and executes it on a compute device.

When either the model or data is large that a single device cannot complete the training in a reasonable amount of time, we resort to ML parallelization approaches that parallelize the computation on distributed devices.

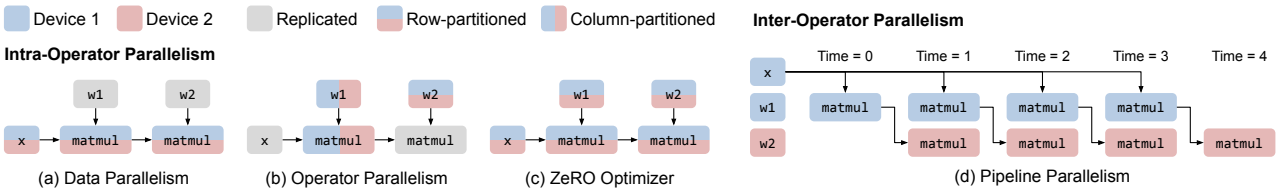


Figure 2: Common parallelization techniques for training a 2-layer Multi-layer Perceptron (MLP). Only the forward pass is shown. "x" is the input data. "w1" and "w2" are two weight matrices.

2.1 Conventional View of ML Parallelism

Existing ML parallelization approaches are typically categorized as data, operator, and pipeline parallelisms.

Data parallelism. In data parallelism, the training data is partitioned across distributed workers, but the model is replicated. Each worker computes the parameter updates on its independent data split, and synchronizes the updates with other workers before the weight update, so that all workers observe consistent model parameters throughout training.

Operator parallelism. When the model is too large to fit in one device, operator parallelism is an effective model parallelism option. Operator parallelism refers to approaches that partition the computation of a specific operator (abbreviated as *op* in the following text), such as matmul shown in Fig. 2b, along *non-batch* axes, and compute each part of the operator in parallel across multiple devices.

Because input tensors are jointly partitioned, when a device computes its op partition, the required portions of input tensors may not reside in its local memory. Communication is thus required to fetch the input data from other devices. When the tensors are partitioned evenly, i.e., SPMD [57], all devices will follow the same collective communication patterns such as all-reduce, all-gather, and all-to-all.

Pipeline parallelism. Instead of partitioning ops, pipeline parallelism places different groups of ops from the model graph, referred as *stages*, on different workers; meanwhile, it splits the training batch as a number of microbatches, and pipelines the forward and backward passes across microbatches on distributed workers, as Fig. 2d shows. Unlike operator parallelism, pipeline parallelism transfers intermediate activations at the forward and backward passes between different workers using point-to-point communication.

Manual combination of parallelisms. Recent development shows the approaches mentioned above need to be combined to scale out today's large DL models [40, 57]. The state-of-the-art training systems, such as Megatron-LM [40, 49], manually design a specialized execution plan that combines these parallelisms for transformer language models, which is also known as *3D Parallelism*. By assuming the model has the same transformer layer repeated, it assigns an equal number of layers to each pipeline stage and applies a hand-designed operator and data parallelism configuration uniformly for all layers. Despite the requirement of strong expertise, the manual plan cannot generalize to different models or different cluster setups (§8.1).

Automatic combination of parallelisms. The configurations of each individual parallelism, their interdependence, and their dependence on model and cluster setups form an intractable space, which prevents the trivial realization of automatically combining these parallelisms. For examples, when coupled with operator parallelism, each time adding a data-parallel replica would require allocating a new set of devices (instead of one single device) as the worker, and figuring out the optimal operator parallelism configurations within those devices. When including pipeline parallelism, the optimal pipelining scheme depends on the data and operator parallelism choices of each pipeline stage and how devices are allocated for each stage. With this conventional view, prior explorations [17, 25, 55, 60] of auto-parallelization are limited to combining data parallelism with at most one model parallelism approach, which misses substantial performance opportunities. We next develop our view of ML parallelisms.

2.2 Intra- and Inter-Operator Parallelisms

Different from the conventional view, in this paper, we recatalog existing parallelization approaches into two orthogonal categories: intra-operator and inter-operator parallelisms. They are distinguished by if they involve partitioning operators along any tensor axis. We next use the examples in Fig. 2 to introduce the two types of parallelisms.

Intra-operator parallelism. An operator works on multi-dimensional tensors. We can partition the tensor along some dimensions, assign the resulting partitioned computations to multiple devices, and let them execute different portions of the operator at the same time. We define all parallelization approaches using this workflow as intra-operator parallelism.

Fig. 2a-c illustrates the application of several typical instantiations of intra-op parallelism on an MLP. Data parallelism [29], by definition, belongs to intra-op parallelism – the input tensors and matmuls are partitioned along the batch dimension, and weight tensors are replicated. Alternatively, when the weights are very large, partitioning the weights (Fig. 2b) leads to the aforementioned operator parallelism adopted in Megatron-LM. Besides operators in the forward or backward passes, one can also partition the operators from the weight update phase, yielding the *weight update sharding* or equivalently the ZeRO [44, 56] technique, commonly comprehended as an optimization of data parallelism.

Due to the partitioning, collective communication is re-

quired at the split and merge of the operator. Hence, a key characteristic of intra-operator parallelism is that it results in substantial communication among distributed devices.

Inter-operator parallelism. We define inter-operator parallelism as the orthogonal class of approaches that *do not* perform operator partitioning, but instead, assign different operators of the graph to execute on distributed devices.

Fig. 2d illustrates the batch-splitting pipeline parallelism as a case of inter-operator parallelism.[2] The pipeline execution can follow different schedules, such as Gpipe [22], PipeDream [38], and synchronous 1F1B [17, 39]. We adopt the synchronous 1F1B schedule throughout this paper as it respects synchronous consistency, and has the same pipeline latency but lower peak memory usage compared to Gpipe.

In inter-operator parallelism, devices communicate only between pipeline stages, typically using point-to-point communication between device pairs. The required communication volume can be much less than the collective communication in intra-operator parallelism. Regardless of the schedule used, due to the data dependency between stages, inter-operator parallelism results in some devices being idle during the forward and backward computation.

By this categorization, the two parallelisms take place at different granularities of the DL computation and have distinct communication requirements, which happen to match the structure of today's compute clusters. We will leverage these properties to design hierarchical algorithms and compilation passes to auto-generate execution plans. Several concurrent work [2, 33, 39, 50] have proposed similar categorization, but Alpa is the first end-to-end system that uses this categorization to automatically generate parallel plans from the full space.

3 Overview

Alpa is a compiler that generates model-parallel execution plans by hierarchically optimizing the plan at two different levels: intra-op and inter-op parallelism. At the intra-op level, Alpa minimizes the cost of executing a stage (i.e., subgraph) of the computational graph, with respect to its intra-operator parallelism plan, on a given device mesh, which is a set of devices that may have high bandwidth between each other (e.g., GPUs within a single server). Different meshes might have different numbers of computing devices according to the workload assigned. At the inter-op level, Alpa minimizes the inter-op parallelization latency, with respect to how to slice the model and device cluster into stages and device meshes and how to map them as stage-mesh pairs. The inter-op optimization depends on knowing the execution cost of each stage-mesh pair reported by the intra-op optimizer. Through this hierarchical optimization process, Alpa generates the execution plan consisting of intra-op and inter-op plans which are locally near-optimal at their respective level of the hierarchy.

To achieve this, Alpa implements three novel compilation passes as Fig. 3 shows. Given a model description, in the form of a Jax [9] intermediate representation (IR), and a cluster configuration, the inter-op compilation pass slices the IR into a number of stages, and slices the device cluster into a number of device meshes. The inter-op pass uses a Dynamic Programming (DP) algorithm to assign stages to meshes and invokes the intra-op compilation pass on each stage-mesh pair, to query the execution cost of this assignment. Once invoked, the intra-op pass optimizes the intra-op parallel execution plan of the stage running on its assigned mesh, by minimizing its execution cost using an Integer Linear Programming (ILP) formulation, and reports the cost back to the inter-op pass. By repeatedly querying the intra-op pass for each allocation of a stage-mesh pair, the inter-op pass uses the DP to minimize the inter-op parallel execution latency and obtains the best slicing scheme of stages and meshes.

Given the output hierarchical plan and a designated pipeline-parallel schedule, each stage is first compiled as a

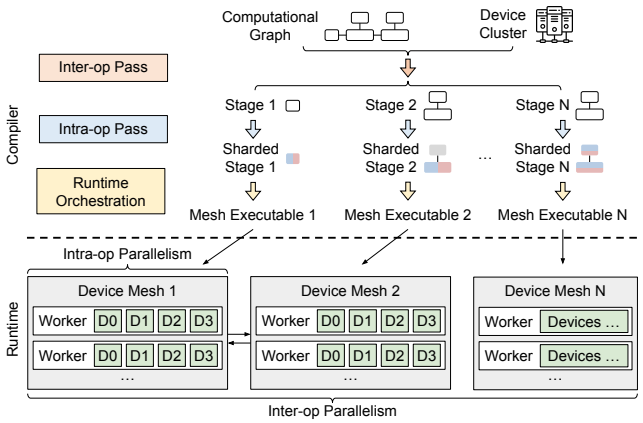


Figure 3: Compiler passes and runtime architecture. A sharded stage is a stage annotated with the sharding specs generated by intra-op pass.

```
# Put @parallelize decorator on top of the Jax functions
@parallelize
def train_step(state, batch):
    def loss_func(params):
        out = state.forward(params, batch["x"])
        return jax.numpy.mean((out - batch["y"]) ** 2)

    grads = grad(loss_func)(state.params)
    new_state = state.apply_gradient(grads)
    return new_state

# A typical training loop
state = create_train_state()
for batch in data_loader:
    state = train_step(state, batch)
```

Figure 4: An example to demonstrate Alpa's API for Jax. The developers uses a Python decorator `@parallelize` to annotate functions that need to be parallelized. The rest of the program is kept intact.

[2]Device placement [36] is another case of inter-op parallelism, which partitions the model graph and executes them on different devices but does not saturate pipelines using multiple microbatches. Hence pipeline parallelism is often seen as a better alternative to it because of less device idle time.

parallel executable on its located mesh. A runtime orchestration pass is invoked to fulfill the communication requirement between two adjacent stages that require communication between the two meshes they locate on. The runtime orchestration pass then generates static instructions specific to each mesh according to the pipeline-parallel schedule and invokes the execution on all meshes.

API. Alpa has a simple API shown in Fig. 4. Alpa requires developers to annotate functions to be parallelized, such as the `train_step()`, using a Python decorator `@parallelize`. Upon the first call to `train_step()`, Alpa traces the whole function to get the model IR, invokes the compilation, and converts the function to a parallel version.

Since the inter-op pass depends on the intra-op pass, in the following text, we first describe the intra-op pass, followed by the inter-op pass, and finally the runtime orchestration pass.

4 Intra-Operator Parallelism

Alpa optimizes the intra-operator parallelism plan within a device mesh. Alpa adopts the SPMD-style intra-op parallelism [31, 57] which partitions operators evenly across devices and executes the same instructions on all devices, as per the fact that devices within a single mesh have equivalent compute capability. This SPMD style significantly reduces the space of intra-op parallelism plans; meanwhile, it conveniently expresses and unifies many important approaches such as data parallelism, ZeRO, Megatron-LM's operator parallelism, and their combinations, which are not fully covered by existing automatic operators parallelism systems, such as Tofu [55] and FlexFlow [25]. Unlike systems that perform randomized search [25] or assume linear graphs [55], Alpa formalizes the problem as an integer linear programming (ILP) and shows it can be solved efficiently for computational graphs with tens of thousands of operators. Next, we describe the space of intra-op parallelism and our solution.

4.1 The Space of Intra-Operator Parallelism

Given an operator in the computational graph, there are multiple possible parallel algorithms to run it on a device mesh. For example, a matrix multiplication $C_{ij} = \sum_k A_{ik} B_{kj}$ corresponds to a three-level for-loop. To parallelize it, we can parallelize the loop i, loop j, loop k, or combinations of them across devices, which would have different computation and communication costs, require different layouts for the input tensors, and result in output tensors with different layouts. If an input tensor does not satisfy the layout requirement, a layout conversion is required, which introduces extra communication costs. The goal of the intra-op pass is to pick one parallel algorithm for every operator to minimize the execution time of the entire graph. Next, we formally define the device mesh and the layout of a tensor and discuss the cost of layout conversion.

Device mesh. A device mesh is a 2-dimensional logical view of a set of physical devices. Each device in the mesh has the same compute capability. Devices can communicate along the first mesh dimension and the second mesh dimension with different bandwidths. We assume different groups of devices along the same mesh dimension have the same communication performance. For a set of physical devices, there can be multiple logical views. For example, given 2 nodes and 8 GPUs per node (i.e., 16 devices in total), we can view them as a 2×8, 1×16, 4×4, 8×2, or 16×1 device mesh. The mapping between physical devices and the logical device mesh view is optimized by the inter-op pass (§5). In the rest of this section, we consider one fixed device mesh view.

Table 1: Sharding specs of a 2-dimentional tensor on a 2×2 device mesh. A is a (N,M) tensor. The device mesh is [[Device 0, Device 1], [Device 2, Device 3]]. Each device stores a partition of A. The first column is the name of the sharding spec. The latter columns use Numpy syntax to describe the partitions stored on each device.

Spec	Device 0	Device 1	Device 2	Device 3
RR	$A[0:N, 0:M]$	$A[0:N, 0:M]$	$A[0:N, 0:M]$	$A[0:N, 0:M]$
S^0S^1	$A[0:\frac{N}{2}, 0:\frac{M}{2}]$	$A[0:\frac{N}{2}, \frac{M}{2}:M]$	$A[\frac{N}{2}:N, 0:\frac{M}{2}]$	$A[\frac{N}{2}:N, \frac{M}{2}:M]$
S^1S^0	$A[0:\frac{N}{2}, 0:\frac{M}{2}]$	$A[\frac{N}{2}:N, 0:\frac{M}{2}]$	$A[0:\frac{N}{2}, \frac{M}{2}:M]$	$A[\frac{N}{2}:N, \frac{M}{2}:M]$
S^0R	$A[0:\frac{N}{2}, 0:M]$	$A[0:\frac{N}{2}, 0:M]$	$A[\frac{N}{2}:N, 0:M]$	$A[\frac{N}{2}:N, 0:M]$
S^1R	$A[0:\frac{N}{2}, 0:M]$	$A[\frac{N}{2}:N, 0:M]$	$A[0:\frac{N}{2}, 0:M]$	$A[\frac{N}{2}:N, 0:M]$
RS^0	$A[0:N, 0:\frac{M}{2}]$	$A[0:N, 0:\frac{M}{2}]$	$A[0:N, \frac{M}{2}:M]$	$A[0:N, \frac{M}{2}:M]$
RS^1	$A[0:N, 0:\frac{M}{2}]$	$A[0:N, \frac{M}{2}:M]$	$A[0:N, 0:\frac{M}{2}]$	$A[0:N, \frac{M}{2}:M]$
$S^{01}R$	$A[0:\frac{N}{4}, 0:M]$	$A[\frac{N}{4}:\frac{N}{2}, 0:M]$	$A[\frac{N}{2}:\frac{3N}{4}, 0:M]$	$A[\frac{3N}{4}:N, 0:M]$
RS^{01}	$A[0:N, 0:\frac{M}{4}]$	$A[0:N, \frac{M}{4}:\frac{M}{2}]$	$A[0:N, \frac{M}{2}:\frac{3M}{4}]$	$A[0:N, \frac{3M}{4}:M]$

Table 2: Several cases of resharding. $all\text{-}gather(x,i)$ means an all-gather of x bytes along the i-th mesh axis. M is the size of the tensor. (n_0, n_1) is the mesh shape.

#	Src Spec	Dst Spec	Communication Cost
1	RR	S^0S^1	0
2	S^0R	RR	$all\text{-}gather(M, 0)$
3	S^0S^1	S^0R	$all\text{-}gather(\frac{M}{n_0}, 1)$
4	S^0R	RS^0	$all\text{-}to\text{-}all(\frac{M}{n_0}, 0)$
5	S^0S^1	$S^{01}R$	$all\text{-}to\text{-}all(\frac{M}{n_0 \cdot n_1}, 1)$

Sharding Spec. We use *sharding spec* to define the layout of a tensor. For an N-dimensional tensor, its sharding spec is defined as $X_0X_1\cdots X_{n-1}$, where $X_i \in \{S, R\}$. If $X_i = S$, it means the i-th axis of the tensor is partitioned. Otherwise, the i-th axis is replicated. For example, for a 2-dimensional tensor (i.e., a matrix), SR means it is row-partitioned, RS means it is column-partitioned, SS means it is both row- and column-partitioned. RR means it is replicated without any partitioning. After we define which tensor axes are partitioned, we then have to map the partitioned tensor axes to mesh axes. We only consider 2-dimensional device meshes, so a partitioned tensor axis can be mapped to either the first or the second axis of the device mesh, or both. We added a superscript to S to denote the device assignment. For example, S^0 means

Table 3: Several parallel algorithms for a batched matmul $C_{b,i,j} = \sum_k A_{b,i,k} B_{b,k,j}$. The notation *all-reduce*(x,i) means an all-reduce of x bytes along the i-th mesh axis. M is the size of the output tensor. (n_0, n_1) is the mesh shape.

#	Parallel Mapping	Output Spec	Input Specs	Communication Cost
1	$i \to 0, j \to 1$	RS^0S^1	RS^0R, RRS^1	0
2	$i \to 0, k \to 1$	RS^0R	RS^0S^1, RS^1R	*all-reduce*$(\frac{M}{n_0}, 1)$
3	$j \to 0, k \to 1$	RRS^0	RRS^1, RS^1S^0	*all-reduce*$(\frac{M}{n_0}, 1)$
4	$b \to 0, i \to 1$	S^0S^1R	S^0S^1R, S^0RR	0
5	$b \to 0, k \to 1$	S^0RR	S^0RS^1, S^0S^1R	*all-reduce*$(\frac{M}{n_0}, 1)$
6	$i \to \{0,1\}$	$RS^{01}R$	$RS^{01}R, RRR$	0
7	$k \to \{0,1\}$	RRR	$RRS^{01}, RS^{01}R$	*all-reduce*$(M, \{0,1\})$

the partitions are along the 0-th axis of the mesh, S^{01} means the partitions take place along both mesh axes. S^0R means the tensor is row-partitioned into two parts – The first part is replicated on device 0 and device 1, and the second part is replicated on device 2 and device 3. Table 1 shows all possible sharding specs of a 2-dimensional tensor on a 2×2 mesh with 4 devices.

Resharding. When an input tensor of an operator does not satisfy the sharding spec of the chosen parallel algorithm for the operator, a layout conversion, namely *resharding*, is required, which might require cross-device communication. Table 2 lists several cases of resharding. For instance, to convert a fully replicated tensor to any other sharding specs (case #1), we can slice the tensor locally without communication; to swap the partitioned axis (case #4), we perform an all-to-all.

Parallel algorithms of an operator. With the definitions above, consider parallelizing a batched matmul $C_{b,i,j} = \sum_k A_{b,i,k} B_{b,k,j}$ on a 2D mesh – Table 3 lists several intra-op parallel algorithms for a batched matmul. Algorithm#1 maps loop i to the 0-th mesh axis and loop j to the 1-th mesh axis, resulting in the output tensor C with a sharding spec RS^0S^1. As the LHS operand $A_{b,i,k}$ and RHS operand $B_{b,k,j}$ both have only one parallelized index, their sharding specs are written as RS^0R and RRS^1, respectively. In this algorithm, each device has all its required input tiles (i.e., a partition of the tensor) stored locally to compute its output tile, so there is no communication cost. In Algorithm #2 in Table 3, when the reduction loop k is parallelized, all-reduce communication is needed to aggregate the partial sum. Similarly, we can derive the sharding specs and communication costs of other parallel algorithms for a batched matmul.

For other primitive operators such as convolution and reduction, we can get a list of possible parallel algorithms following a similar analysis of their math expressions. In the intra-op pass, the model graph is represented in XLA's HLO format [51], which summarizes common DL operators into less than 80 primitive operators, so we can manually enumerate the possible parallel algorithms for every primitive operator.

4.2 ILP Formulation

The total execution cost of a computational graph $G = (V, E)$ is the sum of the compute and communication costs on all nodes $v \in V$ and the resharding costs on all edges $e \in E$. We formulate the cost minimization as an ILP and solve it optimally with an off-the-shelf solver [18].

For node v, the number of possible parallel algorithms is k_v. It then has a communication cost vector c_v of length k_v, or $c_v \in \mathbb{R}^{k_v}$, where c_{vi} is the communication cost of the i-th algorithm. Similarly, node v has a compute cost vector $d_v \in \mathbb{R}^{k_v}$. For each node v, we define an one-hot decision vector $s_v \in \{0,1\}^{k_v}$ to represent the algorithm it uses. $s_{vi} = 1$ means we pick the i-th algorithm for node v. For the resharding cost between node v and node u, we define a resharding cost matrix $R_{vu} \in \mathbb{R}^{k_v \times k_u}$, where R_{vuij} is the resharding cost from the output of i-th strategy of node v to the input of j-th strategy of node u. The objective of the problem is

$$\min_s \sum_{v \in V} s_v^\intercal (c_v + d_v) + \sum_{(v,u) \in E} s_v^\intercal R_{vu} s_u, \tag{1}$$

where the first term is the compute and communication cost of node v, and the second is the resharding cost of the edge (v,u). In this formulation, s is the variable, and the rest are constant values. The term $s_v^\intercal R_{vu} s_u$ in Eq. 1 is quadratic, and cannot be fed into an ILP solver. We linearize [19] the quadratic term by introducing a new decision vector $e_{vu} \in \{0,1\}^{k_v \cdot k_u}$ which represents the resharding decision between node v and u.

Although we can use profiling to get the accurate costs for c_v, d_v, and R_{vu}, we use the following methods to estimate them for simplicity. For communication costs d_v and R_{vu}, we compute the numbers of communicated bytes and divide them by the mesh dimension bandwidth to get the costs. For compute costs c_v, we set all of them as zero following the same motivation in [55]. This is reasonable because: (1) For heavy operators such as matmul, we do not allow replicated computation. All parallel algorithms always evenly divide the work to all devices, so all parallel algorithms of one operator have the same arithmetic complexity; (2) For lightweight operators such as element-wise operators, we allow replicated computation of them, but their computation costs are negligible.

To simplify the graph, we merge computationally-trivial operators, such as element-wise operators, transpose, and reduction, into one of their operands and propagate the sharding spec from the operand. This greatly reduces the number of nodes in the graph, thus the ILP problem size. We do a breath-first-search and compute the depth of each node. The node is merged to the deepest operand.

Once the parallel plan is decided by ILP, we also apply a set of post-ILP communication optimizations, such as replacing all-reduce with reduce-scatter and all-gather, whenever applicable, because the latter reduces the number of replicated tensors and corresponding computations, while keeping the communication volume the same. This achieves the effect of weight update sharding [56] or ZeRO optimizer [44].

5 Inter-Operator Parallelism

In this section, we develop methods to slice the model and device cluster into stage-mesh pairs. Our optimization goal is to minimize the *end-to-end* pipeline execution latency for the entire computational graph. Previous works [17, 33] have considered simplified problems, such as assuming the device for each stage is pre-assigned, and all stages have fixed data or operator parallelism plan. Alpa rids these assumptions by jointly considering device mesh assignment and the existence of varying intra-op parallelism plans on each stage.

5.1 The Space for Inter-Operator Parallelism

Assume the computational graph contains a sequence of operators following the topology order of the graph[3], notated as $o_1, \ldots, o_K$, where the inputs of an operator o_k are from operators $o_1, \ldots, o_{k-1}$. We slice the operators into S stages $s_1, \ldots, s_S$, where each stage s_i consists of operators $(o_{l_i}, \ldots, o_{r_i})$, and we assign each stage s_i to a submesh of size $n_i \times m_i$, sliced from a computer cluster that contains devices, notated as the *cluster mesh* with shape $N \times M$. Let $t_i = t_{intra}(s_i, Mesh(n_i, m_i))$ be the latency of executing stage s_i on a submesh of $n_i \times m_i$, minimized by the ILP and reported back by the intra-op pass (§4). As visualized in Fig. 5, assuming we have B different input microbatches for the pipeline, the total minimum latency[4] for the entire computation graph is written as:

$$T^* = \min_{\substack{s_1,\ldots,s_S; \\ (n_1,m_1),\ldots,(n_S,m_S)}} \left\{ \sum_{i=1}^{S} t_i + (B-1) \cdot \max_{1 \le j \le S} \{t_j\} \right\}. \quad (2)$$

The overall latency contains two terms: the first term is the total latency of all stages, interpreted as the latency of the first microbatch going through the pipeline; the second term is the pipelined execution time for the rest of $B-1$ microbatches, which is bounded by the slowest stage (stage 3 in Fig. 5).

We aim to solve Eq. 2 with two additional constraints: (1) For an operator in the forward pass of the graph, we want to colocate it with its corresponded backward operator on the same submesh. Since backward propagation usually uses the similar set of tensors during forward propagation, this effectively reduces the amount of communication to fetch the required tensors generated at the forward pass to the backward pass. We use the sum of forward and backward latency for t_{intra}, so Eq. 2 reflects the total latency, including both forward and backward propagation. (2) We need the sliced submeshes $(n_1, m_1), \ldots, (n_S, m_S)$ to fully cover the $N \times M$ cluster mesh – we do not waste any compute device resources. We next elaborate on our DP formulation.

[3] We simply use the order of how users define each operator, reflected in the model IR, with the input operator as the origin. This allows us to leverage the inherent locality present in the user's program – closely related nodes in the graph will be more likely to be partitioned into the same stage.

[4] This formulation holds for GPipe and synchronous 1F1B schedules. Other pipeline schedules may require a different formulation.

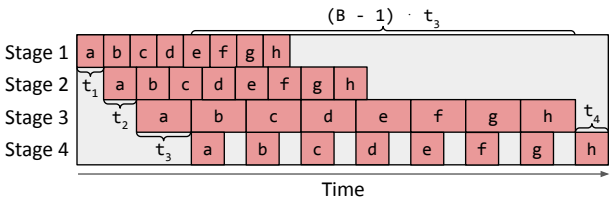


Figure 5: Illustration of the total latency of a pipeline, which is determined by two parts: the total latency of all stages ($t_1 + t_2 + t_3 + t_4$) and the latency of the slowest stage ($(B-1) \cdot t_3$).

5.2 DP Formulation

To ensure all submeshes $(n_1, m_1), \ldots, (n_S, m_S)$ fully cover the $N \times M$ cluster mesh, we reduce the available submesh shapes into two options: (1) one-dimensional submeshes of sizes $(1,1), (1,2), (1,4) \ldots (1,2^m)$ and (2) two-dimensional submeshes of size $(2,M), (3,M), \ldots, (N,M)$ that fully use the second dimension of the cluster mesh (i.e., on a GPU cluster, this means using all compute devices in each physical machine). We include a theorem in Appendix A that proves these submesh shapes can always fully cover the cluster mesh. To assign physical devices in the cluster to the resulting submeshes find by the DP algorithm, we enumerate by assigning devices to larger submeshes first and then to smaller ones. When there are multiple pipeline stages with the same submesh shape, we tend to put neighboring pipeline stages closer on the device mesh to reduce communication latency.

The simplification on submesh shapes works well for most available cloud deep learning setups: On AWS [3], the GPU instances have 1, 2, 4, or 8 GPUs; on GCP [20], the TPU instances have 8, 32, 128, 256 or 512 TPUs. The set of submesh shapes (n,m) excluded by the assumption is with $n > 1$ and $m < M$, which we observe lead to inferior results, since an alternative submesh with shape (n', M) where $n' \cdot M = n \cdot m$ has more devices that can communicate with high bandwidth. With this reduction, we only need to ensure that $\sum_{i=1}^{S} n_i \cdot m_i = N \cdot M$.

To find T^* in Eq. 2, we develop a DP algorithm. The DP first enumerates the second term $t_{max} = \max_{1 \le j \le S} t_j$ and minimizes the first term $t_{total}(t_{max}) = \sum_{1 \le i \le S} t_i$ for each different t_{max}. Specifically, we use the function $F(s,k,d;t_{max})$ to represent the minimal total latency when slicing operators o_k to o_K into s stages and putting them onto d devices so that the latency of each stage is less than t_{max}. We start with $F(0, K+1, 0; t_{max}) = 0$, and derive the optimal substructure of F as

$$F(s,k,d;t_{max}) \quad (3)$$
$$= \min_{\substack{k \le i \le K \\ n_s \cdot m_s \le d}} \left\{ \begin{array}{l} t_{intra}((o_k, \ldots, o_i), Mesh(n_s, m_s), s) \\ + F(s-1, i+1, d - n_s \cdot m_s; t_{max}) \\ \mid t_{intra}((o_k, \ldots, o_i), Mesh(n_s, m_s), s) \le t_{max} \end{array} \right\},$$

and derive the optimal total latency as

$$T^*(t_{max}) = \min_s\{F(s,0,N\cdot M;t_{max})\} + (B-1)\cdot t_{max}. \quad (4)$$

The value of $t_{intra}((o_k,\ldots,o_i),Mesh(n_s,m_s),s)$ is determined by the intra-op pass. It is the lowest latency of executing the subgraph $(o_k,\ldots,o_i)$ on mesh $Mesh(n_s,m_s)$ with s subsequent stages. Note that $Mesh(n_s,m_s)$ is a set of physical devices – hence, we enumerate all the potential choices of logical device mesh shapes (n_l,m_l) satisfying $n_l\cdot m_l = n_s\cdot m_s$. For each choice, we query the intra-op pass with subgraph $(o_k,\ldots,o_i)$, logical mesh (n_l,m_l), and other intra-op options as inputs and get an intra-op plan. We then compile the subgraph with this plan and all other low-level compiler optimizations (e.g., fusion, memory planning) to get an executable for precise profiling. The executable is profiled in order to get the stage latency (t_l) and the memory required on each device to run the stage (mem_{stage}) and to store the intermediate activations (mem_{act}). We check whether the required memory fits the device memory (mem_{device}) according to the chosen pipeline execution schedule. For example, for 1F1B schedule [17, 39], we check

$$mem_{stage} + s\cdot mem_{act} \le mem_{device}. \quad (5)$$

We pick the logical mesh shape that minimizes t_l and fits into the device memory. If none of them fits, we set $t_{intra}=\infty$.

Our algorithm builds on top of that in TeraPipe [33]. However, TeraPipe assumes all pipeline stages are the same, and the goal is to find the optimal way to batch input tokens into micro-batches of different sizes. Instead, Alpa aims to group the operators of a computational graph into different pipeline stages, while assuming the input micro-batches are of the same size. In addition, Alpa optimizes the mesh shape in the DP algorithm for each pipeline stage in inter-op parallelism.

Complexity. Our DP algorithm computes the slicing in $O(K^3NM(N+\log(M)))$ time for a fixed t_{max}. t_{max} has at most $O(K^2(N+\log(M)))$ choices: $t_{intra}((o_i,\ldots,o_j),Mesh(n_s,m_s))$ for $i,j=1,\ldots,K$ and all the submesh choices. The complexity of this DP algorithm is thus $O(K^5NM(N+\log(M))^2)$.

This complexity is not feasible for a large computational graph of more than ten thousand operators. To speed up this DP, we introduce a few practical optimizations.

Performance optimization #1: early pruning. We use one optimization that is similar to that in TeraPipe [33]. We enumerate t_{max} from small to large. When $B\cdot t_{max}$ is larger than the current best T^*, we immediately stop the enumeration. This is because larger t_{max} can no longer provide a better solution. Also, during enumeration of t_{max}, we only evaluate a choice of t_{max} if it is sufficiently larger than the last t_{max} (by at least ε). This allows the gap between the solution found by the DP algorithm and the global optima to be at most $B\cdot\varepsilon$. We empirically choose $\varepsilon = 10^{-6}$ s, and we find that the solution output by our algorithm is the same as the real optimal solution ($\varepsilon = 0$) for all our evaluated settings.

Algorithm 1 Inter-op pass summary.

Input: Model graph G and cluster C with shape (N,M).
Output: The minimal pipeline execution latency T^*.
// Preprocess graph.
$(o_1,\ldots,o_K) \leftarrow$ Flatten(G)
$(l_1,\ldots,l_L) \leftarrow$ OperatorClustering$(o_1,\ldots,o_K)$
// Run the intra-op pass to get costs of different stage-mesh pairs.
submesh_shapes $\leftarrow \{(1,1),(1,2),(1,4),\ldots,(1,M)\} \cup \{(2,M),(3,M),\ldots,(N,M)\}$
for $1\le i\le j\le L$ **do**
  stage $\leftarrow (l_i,\ldots,l_j)$
  for $(n,m)\in$ *submesh_shapes* **do**
    for s **from** 1 **to** L **do**
      $t\_intra(stage,Mesh(n,m),s)\leftarrow\infty$
    end for
    for $(n_l,m_l),opt \in$ LogicalMeshShapeAndIntraOpOptions(n,m) **do**
      $plan \leftarrow$ IntraOpPass$(stage,Mesh(n_l,m_l),opt)$
      $t_l, mem_{stage}, mem_{act} \leftarrow$ Profile$(plan)$
      for s satisfies Eq. 5 **do**
        if $t_l < t\_intra(stage,Mesh(n,m),s)$ **then**
          $t\_intra(stage,Mesh(n,m),s)\leftarrow t_l$
        end if
      end for
    end for
  end for
end for
// Run the inter-op dynamic programming
$T^*\leftarrow\infty$
for $t_{max}\in$ SortedAndFilter(t_intra,ε) **do**
  if $B\cdot t_{max}\ge T^*$ **then**
    break
  end if
  $F(0,L+1,0;t_{max})\leftarrow 0$
  for s **from** 1 **to** L **do**
    for l **from** L **down to** 1 **do**
      for d **from** 1 **to** $N\cdot M$ **do**
        Compute $F(s,l,d;t_{max})$ according to Eq. 3
      end for
    end for
  end for
  $T^*(t_{max})\leftarrow \min_s\{F(s,0,N\cdot M;t_{max})\}+(B-1)\cdot t_{max}$
  if $T^*(t_{max}) < T^*$ **then**
    $T^*\leftarrow T^*(t_{max})$
  end if
end for

Performance optimization #2: operator clustering. Many operators in a computational graph are not computationally intensive (e.g., ReLU), and the exact placement of these operators has little impact on the total execution time. We develop another DP algorithm [4] to cluster neighboring operators to

reduce the total size of the graph Eq. 2 works on. We cluster the operators $(o_1, \ldots, o_K)$ into a series of layers[5] $(l_1, \ldots, l_L)$, where $L \ll K$. The goal of the algorithm is to merge two types of operators: (1) those that do not call for much computation but lengthen the computational graph and (2) neighboring operators that may cause substantial communication if put on different device meshes. We define function $G(k,r)$ as the minimum of maximal amount of data received by a single layer when clustering operators $(o_1, \ldots, o_k)$ into r layers. Note that G has the following optimal substructure:

$$G(k,r) \tag{6}$$
$$= \min_{1 \le i \le k} \left\{ \left| \begin{array}{l} \max\{G(i-1,r-1), C(i,k)\} \\ FLOP(o_i, \ldots, o_k) \le \dfrac{(1+\delta) FLOP_{total}}{L} \end{array} \right. \right\},$$

where $C(i,k)$ denotes the total size of inputs of $(o_i, \ldots, o_k)$ received from $(o_1, \ldots, o_{i-1})$ and $FLOP_{total} = FLOP(o_1, \ldots, o_K)$ is the total FLOP of the whole computational graph. We make sure that each clustered layer's FLOP is within $1+\delta$ times of the average FLOP per layer while minimizing the communication. For the solutions with the same communication cost, we choose the one with the most uniform structure by also minimizing the variance of per-layer FLOP. With our DP algorithm, we can compute the best layer clustering in $O(K^2 L)$ time. Note that L here is a hyperparameter to the algorithm. In practice, we choose a small L based on the number of devices and the number of heavy operators in the graph. We find different choices of L do not affect the final performance significantly.

Alg. 1 summarizes the workflow of the inter-op pass and illustrates its interactions with the intra-op pass in §4.

6 Parallelism Orchestration

After stages, device meshes, and their assignments are decided, at the intra-op level, Alpa compiles each stage against its assigned device mesh, respecting the intra-op parallelism plan output by the ILP solver. The compilation depends on XLA [51] and GSPMD [57], and generates parallel executables for each stage-mesh pair. When needed, the compilation automatically inserts collective communication primitives (see §4) to address the *within-mesh* communication caused by intra-op parallelism.

At the inter-op level, Alpa implements an additional parallelism orchestration pass to address the *cross-mesh* communication between stages, and generate static instructions for inter-op parallel execution.

Cross-mesh resharding. Existing manual systems, such as Megatron-LM [45, 49], constrain all pipeline stages to have the same degrees of data and tensor model parallelism, so the communication between pipeline stages is trivially realized by P2P send/recv between corresponded devices of two equivalent device meshes (Fig. 6a). In Alpa, the device meshes holding two adjacent stages might have different mesh shapes, and the tensor to communicate between two stages might have different sharding specs (Fig. 6b and Fig. 6c). We call this communication pattern as *cross-mesh resharding*, which is a many-to-many multicast problem.

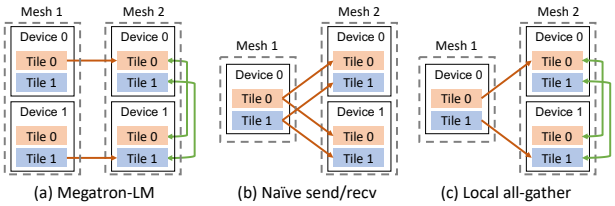


Figure 6: Cross-mesh resharding. Red arrows denote send/recv on slow connections. Green arrows denote all-gather on fast connections. (a) The scatter-gather optimization for equal mesh shapes in Megatron-LM. (b) The naive send/recv for unequal mesh shapes. (c) The generalized local all-gather optimization for unequal mesh shapes.

Given the sharding specs of the tensor on the sender and receiver mesh, Alpa generates a communication plan to address cross-mesh sharding in two iterations. In the first iteration, Alpa calculates the correspondences between tensor partitions (a.k.a. tiles) on the source and destination mesh. Based on that, it generates P2P send/recv primitives between the source devices and destination devices to fulfill the communication. It then takes a second iteration to identify opportunities where the destination tensor has a replication in its sharding spec. In this case, the tensor only needs to be transferred once between two meshes, then exchanged via all-gather across the devices on the destination mesh using its higher bandwidth (Fig. 6) – it rewrites send/recv generated at the first iteration into all-gather to avoid repeated communication.

We call this approach as *local all-gather* cross-mesh resharding. Since the communication between stages is normally small by our design, our experiments show that it performs satisfactorily well (§8.5). We defer the development of the optimal cross-mesh resharding plan to future work.

Generating pipeline execution instructions. As the final step, Alpa generates static execution instructions to launch the training on clusters. Since each stage has different sets of operators and may locate on meshes with different shapes, in contrast to many SPMD pipeline-parallel training systems [40, 57], Alpa adopts an MPMD-style runtime to orchestrate the inter-op parallel execution – Alpa generates distinct static execution instructions for each device mesh.

Alpa develops a set of instructions for inter-op parallel execution, including instructions for allocating and deallocating memory for tensors in a stage, communicating tensors between stages following the cross-mesh resharding plan, synchronization, and computation, etc. According to a user-selected pipeline schedule, Alpa uses a driver process to gen-

[5] Note that the clustering does not exactly reproduce the layers with original machine learning semantics in the model definition.

erate the instructions in advance and dispatches the whole instruction lists to each worker before execution, avoiding driver-worker coordination overheads during runtime.

7 Limitations and Discussion

In this section, we discuss advantages of our view of parallelisms and several limitations of our algorithms.

Compared to existing work that manually combines data, operator, and pipeline parallelism, such as 3D parallelism [45] and PTD-P [40], Alpa's hierarchical view of inter- and intra-op parallelisms significantly advances them with three major flexibility: (1) pipeline stages can contain an uneven number of operators or layers; (2) pipeline stages in Alpa might be mapped to device meshes with different shapes; (3) within each stage, the data and operator parallelism configuration is customized non-uniformly on an operator-by-operator basis. Together, they allow Alpa to unify all existing model parallelism approaches and generalize to model architectures and cluster setups with more heterogeneity.

Despite these advantages, Alpa's optimization algorithms currently have a few limitations:

• Alpa does not model the communication cost between different stages because the cross-stage communication cost is *by nature small*. In fact, modeling the cost in either the DP or ILP is possible, but would require enumerating exponentially more intra-op passes and DP states.

• The inter-op pass currently has a hyperparameter: the number of micro-batches B, which is not optimized by our current formulation but can be searched by enumeration.

• The inter-op pass models pipeline parallelism with a static linear schedule, without considering more dynamic schedules that, for example, parallelize different branches in a computational graph on different devices.

• Alpa does not optimize for the best scheme of overlapping computation and communication; Alpa can only handle static computational graphs with all tensor shapes known at compilation time.

Nevertheless, our results on weak scaling (§8) suggest that Alpa is able to generate near-optimal execution plans for many notable models.

8 Evaluation

Alpa is implemented using about 16K LoC in Python and 6K LoC in C++. Alpa uses Jax as the frontend and XLA as the backend. The compiler passes are implemented on Jax's and XLA's intermediate representation (i.e., Jaxpr and HLO). For the distributed runtime, we use Ray [37] actor to implement the device mesh worker, XLA runtime for executing computation, and NCCL [41] for communication.

We evaluate Alpa on training large-scale models with billions of parameters, including GPT-3 [10], GShard Mixture-of-Experts (MoE) [31], and Wide-ResNet [59]. The testbed is a typical cluster consisting of 8 nodes and 64 GPUs. Each node is an Amazon EC2 p3.16xlarge instance with 8 NVIDIA V100 16 GB GPUs, 64 vCPUs, and 488 GB memory. The 8 GPUs in a node are connected via NVLink. The 8 nodes are launched within one placement group with 25Gbps cross-node bandwidth.

Table 4: Models used in the end-to-end evaluation. LM = language model. IC = image classification.

Model	Task	Batch size	#params (billion)	Precision
GPT-3 [10]	LM	1024	0.35, 1.3, 2.6, 6.7, 15, 39	FP16
GShard MoE [31]	LM	1024	0.38, 1.3, 2.4, 10, 27, 70	FP16
Wide-ResNet [59]	IC	1536	0.25, 1.0, 2.0, 4.0, 6.7, 13	FP32

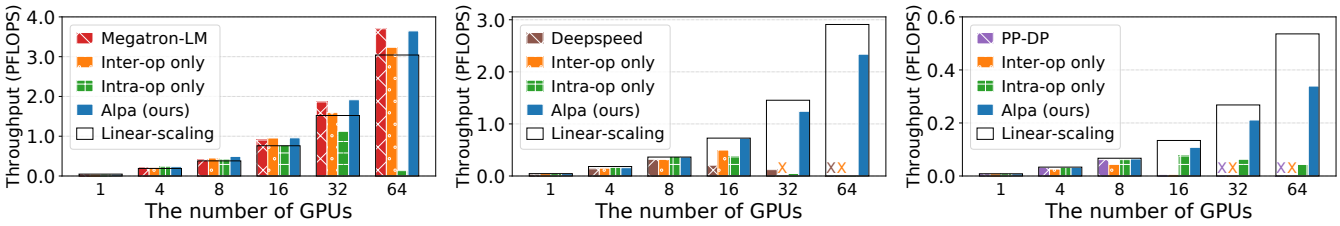
We compare Alpa against two state-of-the-art distributed systems for training large-scale models on GPUs. We then isolate different compilation passes and perform ablation studies of our optimization algorithms. We also include a case study of the execution plans found by Alpa.

8.1 End-to-End Performance

Models and training workloads. We target three types of models listed in Table 4, covering models with both homogeneous and heterogeneous architectures. GPT-3 is a homogeneous transformer-based LM by stacking many transformer layers whose model parallelization plan has been extensively studied [40, 49]. GShard MoE is a mixed dense and sparse LM, where mixture-of-experts layers are used to replace the MLP at the end of a transformer, every two layers. Wide-ResNet is a variant of ResNet with larger channel sizes. It is vastly different from the transformer models and there are no existing manually designed strategies.

To study the ability to train large models, we follow common ML practice to scale the model size along with the number of GPUs, with the parameter range reported in Table 4. More precisely, for GPT-3, we increase the hidden size and the number of layers together with the number of GPUs following [40], whereas for MoE we mainly increase the number of experts suggested by [31, 57]. For Wide-ResNet, we increase the channel size and width factor in convolution layers. For each model, we adopt the suggested global batch size per ML practice [10, 31, 40, 59] to keep the same statistical behavior. We then tune the best microbatch size for each model and system configuration that maximizes the system performance. The gradients are accumulated across microbatches. The detailed model specifications are provided in Appendix B.

Baselines. For each model, we compare Alpa against a strong baseline. We use Megatron-LM v2 [40] as the baseline system for GPT-3. Megatron-LM is the state-of-the-art system for training homogeneous transformer-based LMs on GPUs. It combines data parallelism, pipeline parallelism, and manually-designed operator parallelism (denoted as TMP later). The combination of these techniques is controlled by three integer parameters that specify the parallelism degrees assigned to each technique. We grid-search the three parameters following the guidance of their paper and report the results of the best configuration. Megatron-LM is specialized for GPT-like models, so it does not support other models in Table 4.



(a) GPT (b) MoE (c) Wide-ResNet

Figure 7: End-to-end evaluation results. "×" denotes out-of-memory. Black boxes represent linear scaling.

We use DeepSpeed [45] as the baseline for MoE. DeepSpeed provides a state-of-the-art implementation for training MoE on GPUs. It combines handcrafted operator parallelism for MoE layers and ZeRO-based [44] data parallelism. The combination of these techniques is controlled by several integer parameters that specify the parallelism degree assigned to each technique. We also grid-search them and report the best results. The performance of DeepSpeed on GPT-3 is similar to or worse than Megatron-LM, so we skip it on GPT-3. Note that original GShard-MoE [31] implementation is only available on TPUs, thus we do not include its results, though their strategies [31] are covered by Alpa 's strategy space.

For large Wide-ResNet, there is no specialized system or manually designed plan for it. We use Alpa to build a baseline "PP-DP" whose space only consists of data parallelism and pipeline parallelism, which mimics the parallelism space of PipeDream [38] and Dapple [17].

For all models, we also include the results of using Alpa with only one of intra- and inter-operator parallelism, which mimics the performance of some other auto-parallel systems. The open-source Flexflow [25] does not support the models we evaluate, as it lacks support for many necessary operators (e.g., layer normalization [5], mixed-precision operators). Tofu [55] only supports single node execution and is not open-sourced. Due to both theoretical and practical limitations, we do not include their results and we do not expect Flexflow or Tofu to outperform the state-of-the-art manual baselines in our evaluation.

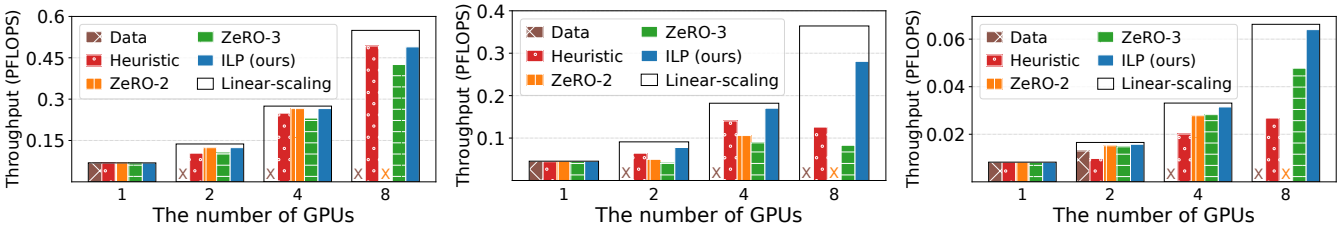
Evaluation metrics. Alpa does not modify the semantics of the synchronous gradient descent algorithm, thus does not affect the model convergence. Therefore, we measure training throughput in our evaluation. We evaluate weak scaling of the system when increasing the model size along with the number of GPUs. Following [40], we use the aggregated peta floating-point operations per second (PFLOPS) of the whole cluster as the metric[6]. We measure it by running a few batches with dummy data after proper warmup. All our results (including those in later sections) have a standard deviation within 0.5%, so we skip the error bars in our figures.

GPT-3 results. The parallelization plan for GPT-3 has been extensively studied [10, 33, 40]. We observe in Fig. 7a that this manual plan with the best grid-searched parameters enables Megatron-LM to achieve super-linear weak scaling on GPT-3. Nevertheless, compared to Megatron-LM, Alpa automatically generates execution plans and even achieves slightly better scaling on several settings. If compared to methods that only use intra-operator parallelism, our results are consistent with recent studies – "Intra-op only" performs poorly on >16 GPUs because even the best plan has to communicate tensors heavily on cross-node connections, making communication a bottleneck. Surprisingly, "Inter-op only" performs well and maintains linear scaling on up to 64 GPUs.

We investigate the grid-searched parameters of the manual plan on Megatron-LM, and compare it to the plan generated by Alpa. It reveals two major findings. First, in Megatron-LM, the best manual plan has TMP as 1, except in rare settings, such as fitting the 39B model on 64 GPUs, where pipeline parallelism alone is unable to fit the model (stage) in GPU memory; meanwhile, data parallelism is maximized whenever memory allows. In practice, gradient accumulation (GA) is turned on to achieve a desired global batch size (e.g., 1024 in our setting). GA amortizes the communication of data parallelism and reduces the bubbles of pipeline parallelism, but the communication of TMP grows linearly with GA steps, which puts TMP disadvantaged. Second, Alpa-generated plan closely resembles the best-performed ones in Megatron-LM, featuring (1) evenly-sized stages, (2) partitioning along the batch dimension in stages when memory is not stressed, but along non-batch dimensions when memory is stressed. One key difference between our plan and the manual plan is that Alpa also partitions the weight update operations when data parallelism exists, which contributes to the slight performance improvement over Megatron-LM. This attributes to the fact that Alpa, as a generic compiler system, can compose a wide range of parallelism approaches, while Megatron-LM, for now, misses weight update sharding support.

MoE results. DeepSpeed adopts a manual operator parallelism plan for MoE models, developed by GShard [31], called *expert parallelism*, which uses a simple rule: it partitions the expert axis for the operators in MoE layers, but switches back to data parallelism for non-expert layers. This expert parallelism is then combined with ZeRO data parallelism and TMP. All of these techniques belong to intra-operator parallelism. Unfortunately, DeepSpeed's specialized implementation does not include any inter-operator parallelism approach, which is

[6] As the models are different for different numbers of GPUs, we cannot measure scaling on the system throughput such as tokens per second or images per second.



(a) GPT (b) MoE (c) Wide-ResNet

Figure 8: Intra-operator parallelism ablation study. "×" denotes out-of-memory. Black boxes represent linear scaling.

required for scaling across multiple nodes with low inter-node bandwidth. Therefore, Deepspeed only maintains a good performance within a node ($\leq$ 8 GPUs) on this cluster. "Intra-op only" fails to scale across multiple nodes due to the same reason. "Inter-op only" runs out of memory on 32 GPUs and 64 GPUs because it is not easy to equally slice the model when the number of GPUs is larger than the number of layers of the model. The imbalanced slicing makes some memory-intensive stages run out of memory.

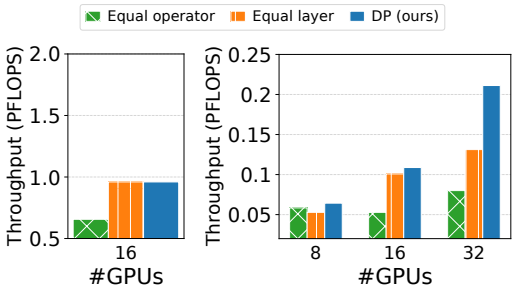
By contrast, Alpa automatically discovers the best execution plans that combine intra- and inter-operator parallelism. For intra-operator parallelism, Alpa finds a strategy similar to expert parallelism and combines it with ZeRO data parallelism, thanks to its ILP-based intra-op pass. Alpa then constructs stages and uses inter-operator parallelism to favor small communication volume on slow connections. Alpa maintains linear scaling on 16 GPUs and scales well to 64 GPUs. Compared to DeepSpeed, Alpa achieves 3.5× speedup on 2 nodes and a 9.7× speedup on 4 nodes.

Wide-ResNet results. Unlike the previous two models that stack the same layer, Wide-ResNet has a more heterogeneous architecture. As the data batch is forwarded through layers, the size of the activation tensor shrinks while the size of the weight tensor inflates. This leads to an imbalanced distribution of memory usage and compute intensity across layers. For this kind of model, it is difficult, if not impossible, to manually design a plan. However, Alpa still achieves a scalable performance on 32 GPUs with 80% scaling. The baselines "PP-DP" and "Inter-op only" run out of memory when training large models, because they cannot partition weights to reduce the memory usage, and it is difficult to construct memory-balanced stages for them. "Intra-only" requires a lot of communication on slow connections, so it cannot scale across multiple nodes. A case study on the generated plan for Wide-ResNet is in §8.6.

8.2 Intra-Op Parallelism Ablation Study

We study the effectiveness of our intra-operator parallelism optimization algorithm. We compare our ILP-based solution against alternatives such as ZeRO optimizer and rule-based partitioning strategies.

Experimental setup. We run a weak scaling benchmark in terms of model size similar to §8.1, but disable pipeline parallelism and gradient accumulation to control variables. The



(a) GPT (b) Wide-ResNet

Figure 9: Inter-operator parallelism ablation study.

benchmark is done on one AWS p3.16xlarge instance with 8 GPUs. In order to simulate an execution environment of large-scale training in one node, we use larger hidden sizes, smaller batch sizes, and smaller numbers of layers, compared to the model configurations in §8.1.

Baselines. We compare automatic solutions for intra-operator parallelism. "Data" is vanilla data parallelism. "ZeRO-2" [44] is a memory-efficient version of data parallelism which partitions gradients and optimizer states. "ZeRO-3" [44] additionally partitions parameters on top of "ZeRO-2". "Heuristic" uses a rule combined with the sharding propagation in GSPMD. It marks the largest dimension of every input tensor as partitioned and runs sharding propagation to get the sharding specs for all nodes in the graph. "ILP" is our solution based on the ILP solver.

Results. As shown in Fig. 8, "Data" runs out of memory quickly and cannot train large models. "ZeRO-2" and "ZeRO-3" resolve the memory problem of data parallelism, but they do not optimize for communication as they always communicate the gradients. When the gradients are much larger than activations, their performance degenerates. "Heuristic" solves the memory issue by partitioning all tensors, but can be slowed down by larger communication. "Auto-sharding" performs best in all cases and maintains a near-linear scaling, because it figures out the correct partition plan that always minimizes the communication overhead.

8.3 Inter-Op Parallelism Ablation Study

We study the effectiveness of our inter-operator parallelism optimization algorithm. We use "DP" to denote our algorithm.

Experimental setup. We report the performance of three variants of our DP algorithm on GPT and Wide-ResNet. The

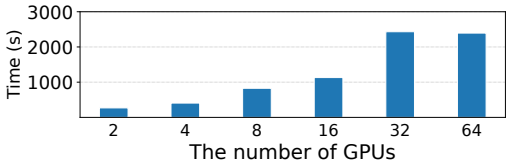


Figure 10: Alpa's compilation time on all GPT models. The model size and #GPUs are simultaneously scaled.

benchmark settings are the same as the settings in §8.1.

Baselines. We compare our DP algorithm with two rule-based algorithms. "Equal operator" disables our DP-based operator clustering but assigns the same number of operators to each cluster. "Equal layer" restricts our DP algorithm to use the same number of layers for all stages.

Results. Fig. 9 shows the result. "DP" always outperforms "Equal operator". This is because "Equal operator" merges operator that should be put onto different device meshes. Alpa's algorithm can cluster operators based on the communication cost and computation balance. Whether "DP" can outperform "Equal layer" depends on the model architecture. On homogeneous models like GPT, the solution of our DP algorithm uses the same number of layers for all stages, so "Equal layer" performs the same as "DP". On Wide-ResNet, the optimal solution can assign different layers to different stages, so "Equal layer" is worse than the full flexible DP algorithm. For Wide-ResNet on 32 GPUs, our algorithm outperforms "Equal operator" and "Equal layer" by 2.6× and 1.6×, respectively.

8.4 Compilation Time

Fig. 10 shows Alpa's compilation time for all the GPT settings in §8.1. The compilation time is a single run of the full Alg. 1 with a provided number of microbatches B. According to the result, Alpa scales to large models or large clusters well, because compilation time grows linearly with the size of the model and the number of GPUs in the cluster. Table 5 reports the compilation time breakdown for the largest GPT model in our evaluation (39B, 64 GPUs). Most of the time is spent on enumerating stage-mesh pairs and profiling them. For the compilation part, we accelerate it by compiling different stages in parallel with distributed workers. For profiling, we accelerate it using a simple cost model built at the XLA instruction level, which estimates the cost of matrix multiplication and communication primitives with a piece-wise linear model. With these optimizations, the compilation and search for a model take at most several hours, which is acceptable as it is much shorter than the actual training time, which can take several weeks.

8.5 Cross-Mesh Resharding

We evaluate our generalized local all-gather optimization for cross-mesh resharding between meshes with different shapes on Wide-ResNet, as shown in Fig. 11. "signal send/recv" is a synthetic case where we only send 1 signal byte between stages, which can be seen as the upper bound of the performance. "w/o local all-gather" disables our local all-gather optimization and uses only send/recv. "w/ local all-gather" enables our local all-gather optimization to move more communication from slow connections to fast local connections, which brings 2.0× speedup on 32 GPUs.

Table 5: Compilation time breakdown of GPT-39B.

Steps	Ours	w/o optimization
Compilation	1582.66 s	> 16hr
Profiling	804.48 s	> 24hr
Stage Construction DP	1.65 s	N/A
Other	4.47 s	N/A
Total	2393.26 s	> 40hr

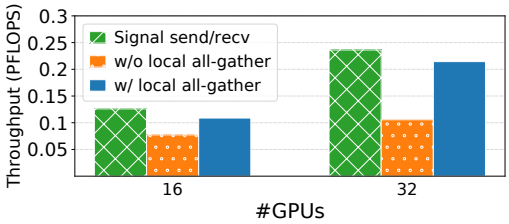


Figure 11: Cross-mesh resharding on Wide-ResNet.

8.6 Case Study: Wide-ResNet

We visualize the parallelization strategies Alpa finds for Wide-ResNet on 16 GPUs in Fig. 12. We also include the visualization of results on 4 and 8 GPUs in Appendix C. On 4 GPUs, Alpa uses only intra-operator parallelism. The intra-operator solution partitions along the batch axis for the first dozens of layers and then switches to partitioning the channel axis for the last few layers. On 16 GPUs, Alpa slices the model into 3 stages and assigns 4, 4, 8 GPUs to stage 1, 2, 3, respectively. Data parallelism is preferred in the first two stages because the activation tensors are larger than weight tensors. In the third stage, the ILP solver finds a non-trivial way of partitioning the convolution operators. The result shows that it can be opaque to manually create such a strategy for a heterogeneous model like Wide-ResNet, even for domain experts.

9 Related Work

Systems for data-parallel training. Horovod [47] and PyTorchDDP [32] are two commonly adopted data-parallel training systems that synchronize gradients using all-reduce. BytePS [26, 43] unifies all-reduce and parameter servers and utilizes heterogeneous resources in data center clusters. AutoDist [60] uses learning-based approaches to compose a data-parallel training strategy. ZeRO [44, 56] improves the memory usage of data parallelism by reducing replicated tensors. MiCS [61] minimizes the communication scale on top of ZeRO for better scalability on the public cloud. In Alpa, data parallelism [27] reduces to a special case of intra-operator parallelism – partitioned along the batch axis.

Systems for model-parallel training. The two major classes of model parallelisms have been discussed in §2. Mesh-

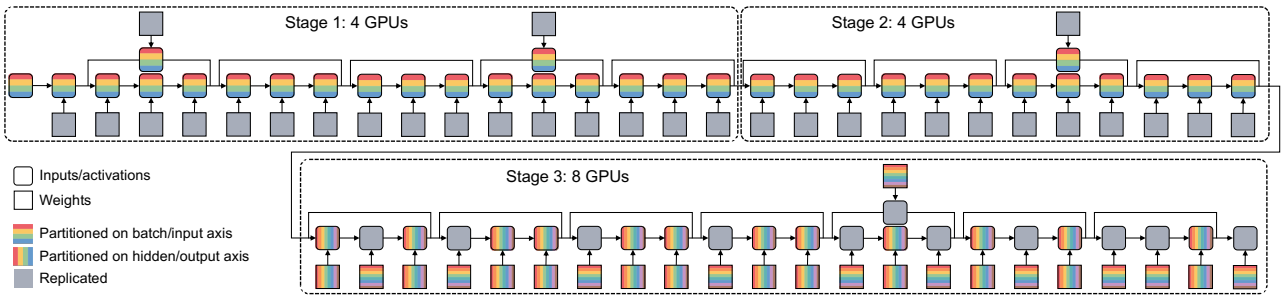


Figure 12: Visualization of the parallel strategy of Wide-ResNet on 16 GPUs. Different colors represent the devices a tensor is distributed on. Grey blocks indicate a tensor is replicated across the devices. The input data and resulting activation of each convolution and dense layer can be partitioned along the batch axis and the hidden axis. The weights can be partitioned along the input and output channel axis.

TensorFlow [48], GSPMD [31,57] and OneFlow [58] provide annotation APIs for users to manualy specifiy the intra-op parallel plan. ColocRL [36] puts disjoint model partitions on different devices *without pipelining*, thereby the concurrency happens only when there exist parallel branches in the model. In contrast, Gpipe [22] splits the input data into micro-batches and forms pipeline parallelisms. PipeDream [38,39] improves GPipe by using asynchronous training algorithms, reducing memory usage, and integrating it with data parallelism. However, PipeDream is asynchronous while Alpa is a synchronous training system. TeraPipe [33] discovers a new pipeline parallelism dimension for transformer-based LMs. Google's Pathway system [7] is a concurrent work of Alpa. Pathway advocates a single controller runtime architecture combining "single program multiple data" (SPMD) and "multiple program multiple data" (MPMD) model. This is similar to Alpa's runtime part, where SPMD is used for intra-op parallelisms and MPMD is used for inter-op parallelism.

Automatic search for model-parallel plans. Another line of work focuses on the automatic discovery of model-parallel training plans. Tofu [55] develops a dynamic programming algorithm to generate the optimal intra-op strategy for *linear* graphs on *a single node*. FlexFlow [25] proposes a "SOAP" formulation and develops an MCMC-based randomized search algorithm. However, it only supports device placement without pipeline parallelism. Its search algorithm cannot scale to large graphs or clusters and does not have optimality guarantees. TensorOpt [11] develops a dynamic programming algorithm to automatically search for intra-op strategies that consider both memory and computation cost. Varuna [2] targets low-bandwidth clusters and focuses on automating pipeline and data parallelism. Piper [50] also finds a parallel strategy with both inter- and intra-op parallelism, but it relies on manually designed intra-op parallelism strategies and analyzes on a uniform network topology and asynchronous pipeline parallel schedules.

Techniques for training large-scale models. In addition to parallelization, there are other complementary techniques for training large-scale models, such as memory optimization [12, 14, 21, 23, 28, 46], communication compression [6, 53], and low-precision training [35]. Alpa can incorporate many of these techniques. For example, Alpa uses rematerialization to reduce memory usage and uses mixed-precision training to accelerate computation.

Compilers for deep learning. Compiler techniques have been introduced to optimize the execution of DL models [13,24,34,51,52,54,62]. Most of them focus on optimizing the computation for a single device. In contrast, Alpa is a compiler that supports a comprehensive space of execution plans for distributed training.

Distributed tensor computation in other domains. Besides deep learning, libraries and compilers for distributed tensor computation have been developed for linear algebra [8] and stencil computations [15]. Unlike Alpa, they do not consider necessary parallelization techniques for DL.

10 Conclusion

We present Alpa, a new architecture for automated model-parallel distributed training, built on top of a new view of machine learning parallelization approaches: intra- and inter-operator parallelisms. Alpa constructs a hierarchical space and uses a set of compilation passes to derive efficient parallel execution plans at each parallelism level. Alpa orchestrates the parallel execution on distributed compute devices on two different granularities. Coming up with an efficient parallelization plan for distributed model-parallel deep learning is historically a labor-intensive task, and we believe Alpa will democratize distributed model-parallel learning and accelerate the adoption of emerging large deep learning models.

11 Acknowledgement

We would like to thank Shibo Wang, Yu Emma Wang, Jinliang Wei, Zhen Zhang, Siyuan Zhuang, anonymous reviewers, and our shepherd, Ken Birman, for their insightful feedback. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba Group, Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[3] AWS Cluster Configuratoins. https://aws.amazon.com/ec2/instance-types/p3/.

[4] Kevin Aydin, MohammadHossein Bateni, and Vahab Mirrokni. Distributed balanced partitioning via linear embedding. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 387–396, 2016.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[6] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 359–375, 2021.

[7] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4, 2022.

[8] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users' guide*. SIAM, 1997.

[9] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[10] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[11] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1967–1981, 2021.

[12] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael W Mahoney, and Joseph E Gonzalez. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *International Conference on Machine Learning*, 2021.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[14] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[15] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed halide. *ACM SIGPLAN Notices*, 51(8):1–12, 2016.

[16] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathy Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. Glam: Efficient scaling of language models with mixture-of-experts, 2021.

[17] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[18] John Forrest and Robin Lougee-Heimer. Cbc user guide. In *Emerging theory, methods, and applications*, pages 257–277. INFORMS, 2005.

[19] Richard J Forrester and Noah Hunt-Isaak. Computational comparison of exact solution methods for 0-1 quadratic programs: Recommendations for practitioners. *Journal of Applied Mathematics*, 2020, 2020.

[20] Google Clould TPU Cluster Configurations. https://cloud.google.com/tpu.

[21] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[22] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32:103–112, 2019.

[23] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *arXiv preprint arXiv:1910.02653*, 2019.

[24] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[25] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[26] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479, 2020.

[27] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[28] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616*, 2020.

[29] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[30] Woo-Yeon Lee, Yunseong Lee, Joo Seong Jeong, Gyeong-In Yu, Joo Yeon Kim, Ho Jin Park, Beomyeol Jeon, Wonwook Song, Gunhee Kim, Markus Weimer, et al. Automating system configuration of distributed machine learning. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2057–2067. IEEE, 2019.

[31] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[32] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[33] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. *arXiv preprint arXiv:2102.07988*, 2021.

[34] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[35] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[36] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

[37] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging ai applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[38] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[39] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[40] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[41] NVIDIA. The nvidia collective communication library, 2018.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[43] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[44] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[45] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[46] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840*, 2021.

[47] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[48] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *arXiv preprint arXiv:1811.02084*, 2018.

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[50] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34, 2021.

[51] Google XLA Team. Xla: Optimizing compiler for machine learning, 2017.

[52] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[53] Thijs Vogels, Sai Praneeth Karinireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances In Neural Information Processing Systems 32 (Nips 2019)*, 32(CONF), 2019.

[54] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. Pet: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.

[55] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[56] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. Automatic cross-replica sharding of weight update in data-parallel training. *arXiv preprint arXiv:2004.13336*, 2020.

[57] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: General and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[58] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. Oneflow: Redesign the distributed

deep learning framework from scratch. *arXiv preprint arXiv:2110.15032*, 2021.

[59] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[60] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. Autosync: Learning to synchronize for data-parallel distributed deep learning. *Advances in Neural Information Processing Systems*, 33, 2020.

[61] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. Mics: Near-linear scaling for training gigantic model on public cloud. *arXiv preprint arXiv:2205.00119*, 2022.

[62] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.

A Proof of Submesh Shape Covering

We prove the following theorem which shows we can always find a solution that fully covers the cluster mesh (N,M) with our selected submesh shapes in §5.2: (1) one-dimensional submeshes of shape $(1,1),(1,2),(1,4)\dots(1,2^m)$ where $2^m = M$ and (2) two-dimensional submeshes of shape $(2,M),(3,M),\dots,(N,M)$.

Theorem 1. *For a list of submesh shapes $(n_1,m_1),\dots(n_S,m_S)$, if $\sum_i n_i \cdot m_i = N \cdot M$ and each (n_i,m_i) satisfies either (1) $n_i = 1$ and $m_i = 2^{p_i}$ is a power of 2 or (2) $m_i = M$, then we can always cover the full (N,M) mesh where $M = 2^m$ with these submesh shapes.*

Proof. We start with putting the second type submesh into the full mesh. In this case, because $m_i = M$, these submeshes can cover the full second dimension of the full mesh. After putting all the second kind of submeshes into the mesh, we reduce the problem to fit a cluster mesh of shape (N,M) with submeshes with shape $(1,2^{p_1}),\dots,(1,2^{p_S})$ where all $p_i \in \{0,1,\dots,m-1\}$. Note that now we have

$$2^{p_1} + \cdots + 2^{p_S} = N \cdot 2^m. \tag{7}$$

We start an induction on m. When $m = 1$, we have all $p_i = 0$ and thus all the submeshes are of shape $(1,1)$, which means that all the submeshes can definitely cover the full mesh. Assume the above hold for all $m = 1,2,\dots,k-1$. When $m = k$, note that in this case the number of submeshes with $p_i = 0$ should be an even number, because otherwise the left hand side of Eq. 7 will be an odd number while the right hand side is always an even number. Then we can split all submeshes with shape $p_i = 0$ into pairs, and we co-locate each pair to form a $(1,2)$ mesh. After this transformation, we have all $p_i > 0$, so we can subtract all p_i and m by 1 and reduce to $m = k-1$ case. Therefore, the theorem holds by induction. □

B Model Specifications

For GPT-3 models, we use sequence length = 1024 and vocabulary size = 51200 for all models. Other parameters of the models are listed in Table. 6. The last column is the number of GPUs used to train the corresponding model.

For GShard MoE models, we use sequence length = 1024 and vocabulary size = 32000 for all models. Other parameters of the models are listed in Table. 7. The last column is the number of GPUs used to train the corresponding model.

For Wide-ResNet models, we use input image size = (224, 224, 3) and #class = 1024 for all models. Other parameters of the models are listed in Table. 8. The last column is the number of GPUs used to train the corresponding model.

C Extra Case Study

We visualize the parallelization strategies Alpa finds for Wide-ResNet on 4 and 8 GPUs in Fig. 13.

Table 6: GPT-3 Model Specification

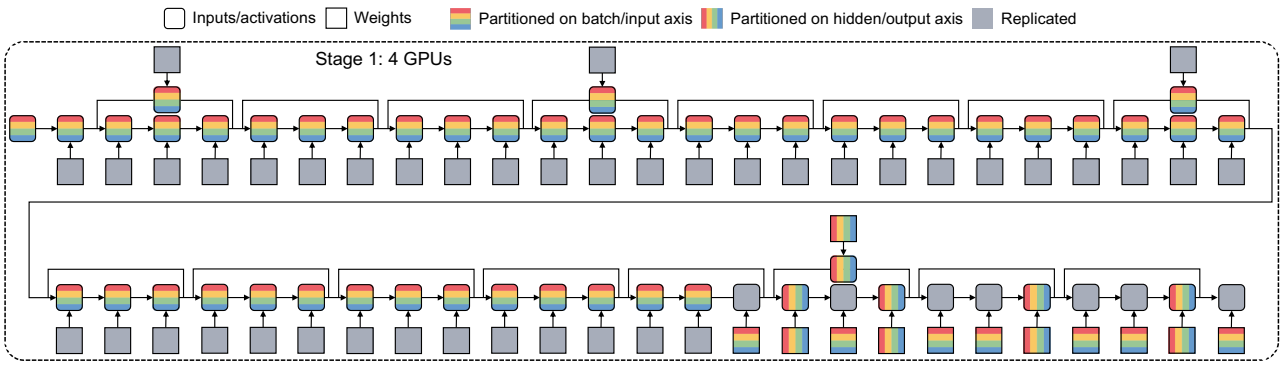
#params	Hidden size	#layers	#heads	#gpus
350M	1024	24	16	1
1.3B	2048	24	32	4
2.6B	2560	32	32	8
6.7B	4096	32	32	16
15B	5120	48	32	32
39B	8192	48	64	64

Table 7: GShard MoE Model Specification

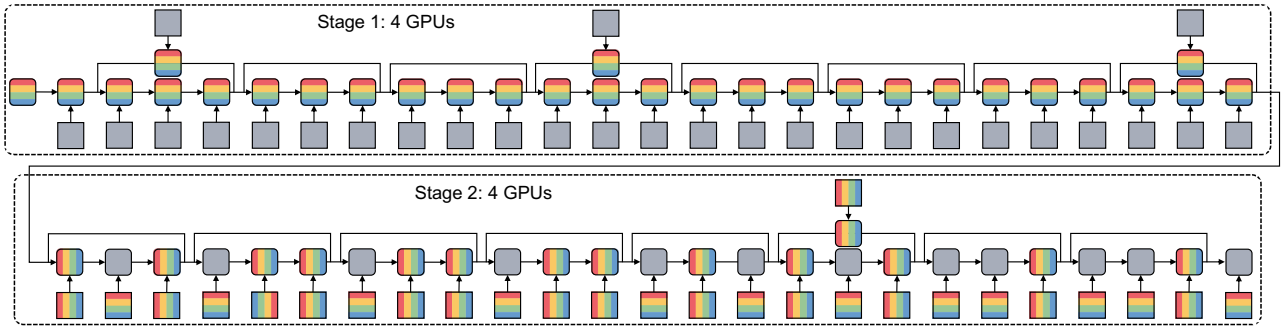
#params	Hidden size	#layers	#heads	#experts	#gpus
380M	768	8	16	8	1
1.3B	768	16	16	16	4
2.4B	1024	16	16	16	8
10B	1536	16	16	32	16
27B	2048	16	32	48	32
70B	2048	32	32	64	64

Table 8: Wide-ResNet Model Specification

#params	#layers	Base channel	Width factor	#gpus
250M	50	160	2	1
1B	50	320	2	4
2B	50	448	2	8
4B	50	640	2	16
6.8B	50	320	16	32
13B	101	320	16	64



(a) Parallel strategy of Wide-ResNet on 4 GPUs.



(b) Parallel strategy of Wide-ResNet on 8 GPUs.

Figure 13: Visualization of the parallel strategy of Wide-ResNet on 4 and 8 GPUs. Different colors represent the devices a tensor is distributed on. Grey blocks indicate a tensor is replicated across all devices. The input data and resulting activation of each convolution or dense layer can be partitioned along the batch axis and the hidden axis. The weights can be partitioned along the input and output channel axis.

Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters

Jayashree Mohan[⋆*], Amar Phanishayee[⋆], Janardhan Kulkarni[⋆], Vijay Chidambaram[†‡]
[⋆]*Microsoft Research* [†]*University of Texas at Austin* [‡]*VMware Research*

Abstract

Training Deep Neural Networks (DNNs) is a popular workload in both enterprises and cloud data centers. Existing schedulers for DNN training consider GPU as the dominant resource and allocate other resources such as CPU and memory proportional to the number of GPUs requested by the job. Unfortunately, these schedulers do not consider the impact of a job's sensitivity to allocation of CPU and memory resources. In this work, we propose Synergy, a resource-sensitive scheduler for shared GPU clusters. Synergy infers the sensitivity of DNNs to different resources using optimistic profiling; some jobs might benefit from more than the GPU-proportional allocation and some jobs might not be affected by less than GPU-proportional allocation. Synergy performs such multi-resource workload-aware assignments across a set of jobs scheduled on shared multi-tenant clusters using a new near-optimal online algorithm. Our experiments show that workload-aware CPU and memory allocations can improve average job completion time by upto 3.4×, by better utilizing existing cluster resources, compared to traditional GPU-proportional scheduling.

1 Introduction

The widespread popularity of Deep Neural Networks (DNNs) makes training such models an important workload in both enterprises and cloud data centers. Training a DNN is resource-intensive and time-consuming. Enterprises typically setup large multi-tenant clusters, with expensive hardware accelerators like GPUs, to be shared by several users and production groups [31, 56]. In addition to the model-specific parameters and scripts, jobs specify their GPU demand before being scheduled to run on available servers. Jobs are scheduled and managed either using traditional big-data schedulers, such as Kubernetes [10] or YARN [51], or using modern schedulers that exploit DNN job characteristics for better performance and utilization [11, 26, 33, 35, 42, 46, 55]. These DNN schedulers decide how to allocate GPU resources to many jobs while implementing complex cluster-wide scheduling policies to optimize for objectives such as average job completion times (JCT), makespan, or user-level fairness.

*Work done as a MSR intern in Project Fiddle.

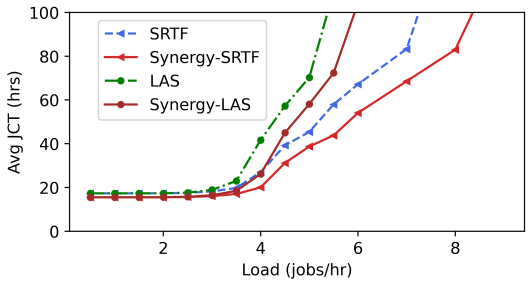


Figure 1: **Average JCT with Synergy**. Synergy is able to significantly reduce average JCT and support higher load for different scheduling policies (shown here on a cluster of 128 GPUs for a Philly-derived trace as we vary load [5]).

Current DNN cluster schedulers assume GPUs to be the dominant resource in the scheduling task [11, 26, 31, 33, 35, 42, 46, 55]; i.e., a user requests a fixed number of GPUs for her DNN job, and when the requested number of GPUs are all available, the job is scheduled to run. Other resources such as CPU and memory are allocated proportional to the number of GPUs assigned to the job (*GPU-proportional* allocation).

However, we identify an important property of DNN training jobs that GPU-proportional allocation is unable to exploit: DNNs exhibit varied sensitivity to the amount of auxiliary resources like CPU and memory allocated to the job. Prior work has shown that ingesting data for ML training jobs, i.e., reading data from storage to memory, and pre-processing them at the CPU is computationally expensive, thereby resulting in *data stalls* in both research [39] and industry scale training at large enterprises such as Google [40] and Facebook [59]. For instance, some image and video recognition models achieve up to 3× speedup by overcoming *data stalls* (§2) when the CPUs allocated exceed their GPU-proportional share, while other models like GNMT are unaffected when the CPUs assigned are less than GPU-proportional share.

Our main insight here is that allocating these auxiliary resources in a workload-aware fashion, rather than the traditional GPU-proportional allocation can significantly improve performance by effectively utilizing *cluster-wide* resources. Based on this insight, we propose *Synergy*, a resource-sensitive scheduler for homogeneous, multi-tenant GPU clusters. Figure 1 shows the average job completion time (JCT)

in the cluster as we vary load, for two scheduling policies; Synergy's resource-sensitive allocation is able to significantly improve average JCT in the cluster and sustain a higher load compared to GPU-proportional allocation.

Synergy profiles the sensitivity of DNNs to auxiliary resources and allocates them disproportionately among jobs rather than using traditional GPU-proportional allocation. While doing so, Synergy ensures that a job gets less than GPU-proportional auxiliary resources *only* if such an allocation does not degrade the job throughput compared to a GPU-proportional allocation. Such allocation enables Synergy to mitigate data stalls in several models, thereby significantly increasing the overall cluster throughput.

Efficiently exploiting the heterogeneity in resource sensitivity among DNN jobs raises two important problems which have not been tackled by prior work:

- What is the ideal resource requirement for each job (with fixed GPU demand) and how can this be determined with low overhead?
- How should we pack these jobs onto servers along multiple resource dimensions efficiently, especially when we can tune the job's demand for these resources?

Optimistic profiling. Synergy exploits the predictability of DNN computation to measure the job throughput as we vary the amount of CPU and memory allocated to the job. This is performed offline by the Synergy scheduler, prior to job execution on the cluster. However, profiling all possible combinations of CPU, and memory values is computationally expensive. Therefore, Synergy introduces optimistic profiling; it empirically profiles the job throughput for varying CPU allocations, assuming maximum memory allocation. It then *analytically* estimates the job throughput for all combinations of CPU and memory.A key insight that makes such analytical modelling feasible is the predictable nature of job performance to memory allocation when using DNN-aware caching like MinIO [39] that guarantees a certain cache hit rate. We show in §3.1 that our optimistically profiled model performance closely resembles the true empirical values, while significantly reducing profiling time (by up to 30×). Using these profiles, Synergy identifies the best resource allocation beyond which the job throughput has diminishing returns.

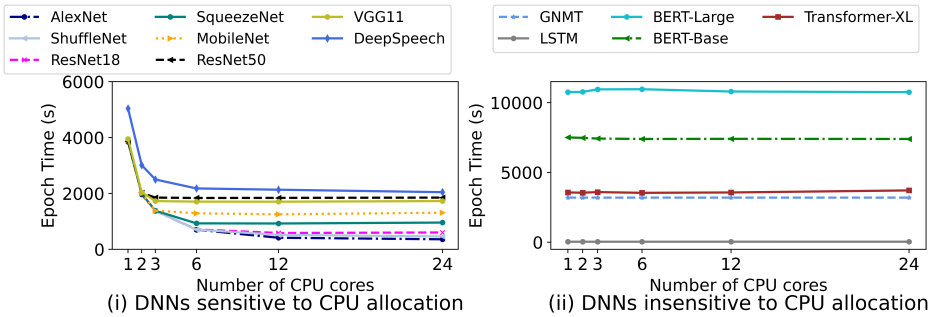
Scheduling mechanism. Synergy makes a round-based scheduling decision similar to prior DNN schedulers [42]. In each round (say 5 minutes), we identify the set of jobs that are runnable in the cluster using a scheduling policy such as FIFO [51, 57], SRTF [12], LAS [26, 43], FTF [35], etc. Synergy's scheduling mechanism then packs these jobs among available servers in the cluster along all resource dimensions identified in the profiling phase. This is analogous to multi-dimensional bin-packing problem, which is NP-Hard [53], and hence requires approximate solutions. But unlike prior work in big-data scheduling which tackles the problem of multi-dimensional bin-packing with fixed resource demands (for *e.g.,* Tetris [23], DRF [21]), Synergy has to contend with fungible resource demands. This introduces two challenges that need to be solved in tandem: First to find an optimal partition of CPU and memory among jobs to maximize throughput while ensuring fair allocations (every job's throughput is at least that of GPU-proportional allocation), and second, a feasible packing of these resources among jobs.

In this paper, we propose two effective algorithms to enable such fungible multi-dimensional bin-packing. Our first algorithm, Synergy-OPT, is formulated as a linear program and enables determining an upper-bound on achievable throughput by an optimal solution for a given workload trace. However, we find that Synergy-OPT is impractical for two reasons: (1) it is computationally expensive as we scale cluster size, and (2) it produces fractional GPU allocations that cannot be achieved in real deployments. Nevertheless, its solution provides an aspirational optimal goal that we can use to measure the efficacy of any practical solution. The second algorithm, Synergy-TUNE, is fast and near-optimal (within 10% of Synergy-OPT in evaluation). If a job to be scheduled does not fit in the cluster along all the resource dimensions, we revert the job demands to GPU-proportional if its current demands are above it. If the job's demands are already GPU-proportional or below, then we find a suitable job in the cluster with higher than GPU-proportional allocation, which is then reverted to GPU-proportional. Synergy-TUNE also outperforms simpler greedy approaches (Synergy-GREEDY) that recursively pack jobs along multiple resource dimensions using a first-fit allocation strategy [20].

We implement a prototype of Synergy and an accompanying event-driven simulator in Python. Synergy transparently communicates with the DNN job using a thin iterator API, that is a wrapper around the existing data iterator, thereby requiring minimal code changes to the DNN job script. Across various scheduling policies, and workload traces, we show that Synergy improves cluster objectives such as average JCT by up to 1.5× on a physical cluster of 32 GPUs. On a large simulated cluster of up to 512 GPUs, Synergy improves average JCT by up to 3.4×. Synergy is open sourced at https://github.com/msr-fiddle/synergy.

In summary, our paper makes the following contributions.

- We identify the importance and need for resource-sensitive scheduling of DNN jobs in multi-tenant GPU clusters (§2).
- We present Synergy, a resource-sensitivity aware scheduler that optimistically profiles the job's resource demands and performs disproportionate allocations such that no job achieves lower than GPU-proportional throughput (§3).
- We present a heuristic scheduling mechanism Synergy-TUNE, that maps the allocations calculated by the profiler onto the cluster, while better utilizing the resources compared to a GPU-proportional allocation (§4).
- In extensive experimentation on physical and simulated clusters, Synergy's techniques improve average JCT by up to 3.4×, thus supporting a higher input load (§5).



(a) CPU sensitivity

CPU: GPU	SKU
3:1	NVIDIA DGX-2 Internal servers at X
4:1	AWS p3.16xlarge
5:1	NVIDIA DGX-1 Azure NDv2
6:1	Azure NC24s_v3

(b) GPU VM SKUs

Figure 2: **CPU sensitivity**. This graph plots the epoch time for DNNs as we vary the CPU:GPU ratio for single-GPU training. Some jobs such as Transformers need as few as 1 CPU core per GPU to achieve maximum training speed; others like ShuffleNet need more than 12 CPU cores per GPU to eliminate data stalls. State-of-the-art GPU VMs have a CPU:GPU ratio as few as 3.

2 Background and Motivation

In this section, we briefly describe DNN scheduling, introduce the terminology used in the rest of the paper, and motivate resource-sensitive DNN cluster scheduling.

Scheduling ML training jobs in a cluster. Training a ML model is a resource intensive and long-running task (order of hours to days). Collocating ML training workloads in a shared, multi-tenant cluster is a very common setup in several large organizations, for both research and production [26, 35, 42, 46, 55]. Our work targets state-of-the-art multi-tenant clusters similar to the ones published by prior large-scale studies by organizations like Microsoft [31] and Alibaba [56]. These clusters use on-premise servers or cloud VMs with pre-defined GPU, CPU, and memory resources. The cluster itself is shared by multiple users and jobs, and each server can host more than one job each with varying resource usage (some heavy on CPU side pre-processing, while others heavy on GPU computation). For example, a server with 8 GPUs can host 8 single-GPU jobs from different users.

Scheduling policy and mechanism. When jobs are submitted to a scheduler, a scheduling policy such as First In, First Out (FIFO) [51, 57], Shortest Remaining Time First (SRTF) [12], Least Attained Service (LAS) [26, 43], or Finish Time Fairness (FTF) [35] decides the set of jobs (J) to be run on the cluster. A scheduling mechanism then identifies where job J should be run, and how much resources to allocate to the job. The GPU demand for a job is fixed (requested by the user), while the CPU and memory allocation is fungible.

GPU-proportional allocation. During DNN training, a mini-batch of data is first fetched from storage to memory, where it is cached for subsequent accesses. It is then pre-processed at the CPU, and then copied over to the GPU for processing. Existing DNN schedulers [26, 35, 42, 55], and those used in real-world GPU clusters [5, 31], including recent schedulers that offer GPU elasticity [30, 48], all allocate CPU and memory resources to a job using a GPU-proportional allocation. For instance, consider a server with 4 GPUs, 16 CPUs and 200 GB memory. If a job requests 1 GPU, then it is allocated 4 CPUs and 50GB memory.

2.1 Motivation : Resource sensitivity

Insight. The main insight that motivates our work is that DNNs co-scheduled on a cluster exhibit different levels of sensitivity to CPU and memory allocations during training. Therefore, it is possible to improve the overall cluster utilization and efficiency by performing resource-sensitive allocations instead of the ubiquitously used GPU-proportional allocation. Prior work on characterization study of jobs in Microsoft's Philly cluster [31] shows that CPU cycles are under-utilized in multi-tenant clusters; we use this as motivation to show that we can *exploit the disparity in resource requirements* ***across jobs*** *to improve overall cluster utilization without any hardware upgrades (storage, CPU, or memory).*

Figure 2a plots the per-epoch time for various DNNs when trained on a single GPU by varying the number of CPUs allocated to the job (ensuring that the dataset is fully cached for each job). Figure 2a(i) shows that most image and speech models are sensitive to CPU allocations; smaller models like ShuffleNet and ResNet18 require 9–24 CPU cores per GPU to pre-process data items. However, state-of-the-art ML optimized servers and cloud GPU VMs have a CPU:GPU ratio as few as 3 as shown in Table 2b [1–3, 6, 18, 34]. Increasing the CPU:GPU ratio from 3 to 12 results in 3.1× faster training for AlexNet, and increasing it to 9 results in 2.3× faster training for ResNet18. On the other hand, most language models are insensitive to CPU allocations as shown in Figure 2a(ii). This is because they have modest input data pre-processing requirements. Transformer models for example, unlike image classification models, do not perform several unique data augmentation operations for each data item in every epoch [39].

Next, to understand the importance of memory alloca-

Job	Model
J_1	ResNet18
J_2	Audio-M5
J_3	Transformer
J_4	GNMT

Table 1: Example jobs

Server	Job	GPU	CPU	Mem
S_1	$J1$	4	12	250
	$J2$	4	12	250
S_2	$J3$	4	12	250
	$J4$	4	12	250

Table 2: GPU-proportional allocation

Server	Job	GPU	CPU	Mem
S_1	$J1$	4	23	400
	$J3$	4	1	100
S_2	$J2$	4	12	450
	$J4$	4	12	50

Table 3: Resource-sensitive allocation

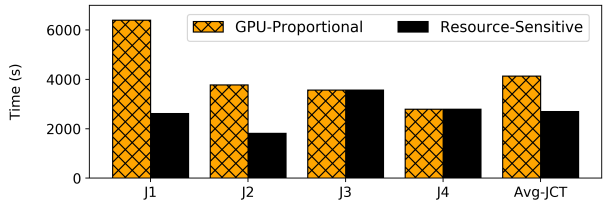


Figure 3: **Resource sensitive scheduling**. We compare the runtime of the jobs with two different schedules; GPU-proportional and resource-sensitive. By allocating resources disproportionately, CPU and memory sensitive jobs see increased throughputs which reduces the average JCT by 1.5×.

tions, we train two models; an image classification model - ResNet18 on OpenImages [22] and a language model GNMT on WMT, with varying memory allocations on a server whose GPU-proportional share of memory per GPU is 62GB. We observe that GNMT is insensitive to memory allocation; even if only 20GB memory is allocated (which is the required process memory for training), the training throughput is unaffected. However, increasing the memory from 62GB (GPU-proportional allocation) to 500GB (max) for ResNet18 speeds up training by almost 2×. This is because, language models like GNMT, and transformers are GPU compute bound. Therefore, fetching data items from storage if they are not available in memory does not affect training throughput. On the other hand, image and speech models benefit from larger DRAM caches. If a data item is not cached, the cost of fetching it from the storage device can introduce fetch stalls in training [39, 40, 59].

Takeaway. *When two jobs have to be scheduled on the same server, it is possible to co-locate a CPU-sensitive job with a CPU-insensitive one. This allows CPU allocation to be performed in a resource-sensitive manner rather than GPU-proportional allocation. Similarly, it is always beneficial to pack a memory-sensitive job with an insensitive one, allowing disproportionate resource-sensitive sharing of memory to improve the aggregate cluster throughput.*

Example. We now show how resource-sensitivity-aware scheduling can improve cluster efficiency using a simple example. We run the experiment on two physical servers each with 8 GPUs, 24 CPUs and 500GB DRAM (internal servers at a large cloud provider X). Let's say we have 4 jobs in the scheduling queue, each requesting 4 GPUs as shown in Table 1. We consider two different schedules; (1) GPU-proportional allocation and (2) resource-sensitive allocation. The results of these schedules are shown in Table 2 and Table 3. Figure 3 compares the epoch time of each of these jobs in the two scenarios. The increased resource allocation to CPU and memory sensitive jobs in Schedule 2 speeds up $J1$ and $J2$ significantly, while leaving the runtime of $J3$ and $J4$ unaffected. The average JCT in the cluster thus drops by 1.5× due to resource-sensitive allocations.

2.2 Synergy Scheduling Policies

Synergy is not constrained to one particular scheduling policy, but is instead general enough to improve a wide range of scheduling policies (*e.g.,* LAS, FIFO, SRTF, FTF, etc), creating Synergy-augmented variants for all of them. The main challenge that Synergy addresses is, finding an efficient partition of available cluster CPU and memory among jobs to maximize throughput while ensuring that every job's throughput is at least that of GPU-proportional allocation. Synergy 's innovation thus lies in exploiting the differences in resource sensitivity across jobs to improve overall cluster metrics.

2.3 Assumptions & Limitations

In the context of this work, we explicitly highlight certain practical assumptions, many of which are derived directly from large multi-tenant clusters we analyze - homogeneous clusters, fixed GPU allocation for the lifetime of a job, and the use of MinIO cache. Synergy 's design is not tied to these assumptions, but it aids in focused profiling (reducing the dimensionality of the search space). In a large scale, multi-tenant, production cluster, it is practical to assume that there are tens of thousands of accelerators per homogeneous cluster, and the GPU allocation for a job remains constant. While recent works explore scheduling DNN jobs in heterogeneous clusters [11, 33, 42], and GPU elasticity [48], there are several practical challenges in seamlessly supporting these features. For instance, with elastic training, the impact of changing batch sizes and hyperparameters on training accuracy is unclear for a wide variety of tasks. We provide a detailed discussion on the practicality of each of these assumptions made by Synergy, and what it means to relax these assumptions for Synergy in Section 6.

3 Synergy: Design

Overview. Synergy is a round-based scheduler that arbitrates multi-dimensional resources (GPU, CPU, and memory) in a

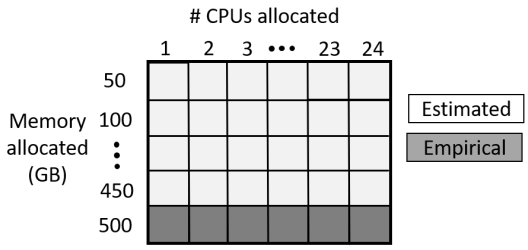


Figure 4: **Optimistic profiling** empirically evaluates the sensitivity of a model to varying # CPUs assuming a fully cached dataset; the rest of the matrix is completed using estimation

homogeneous cluster. Synergy augments existing scheduling policies with *resource sensitivity* in two steps. First, it identifies the job's best-case CPU and memory requirements using *optimistic profiling* (§3.1). Synergy then identifies a set of runnable jobs for the given round using a scheduling policy (*e.g.,* SRTF, FTF, LAS, etc) such that their collective GPU demand is less than or equal to the GPUs available in the cluster. Then, using the profiled resource demands, Synergy packs these jobs on to the available servers along multiple resource dimensions using a near-optimal heuristic algorithm (§4). At the end of a round, the set of runnable jobs are updated using the scheduling policy, and their placement decisions are recomputed. We now discuss both the components of Synergy in detail. Note that Synergy only alters the auxiliary resource allocations; GPU demands are left unaltered for the lifetime of a job and are provided as inputs by the user.

3.1 Optimistic Profiling

A DNN job is profiled for its resource sensitivity once per lifetime of the job, i.e. on job arrival. Each incoming job is profiled by varying the CPU and memory allocated to the job. A *resource sensitivity matrix* is then constructed for discrete combinations of CPU and memory allocations as shown in Figure 4. Since DNN training has a highly predictable structure, empirically evaluating training throughput for a few iterations gives a fair estimate of the actual job throughput [39, 55].

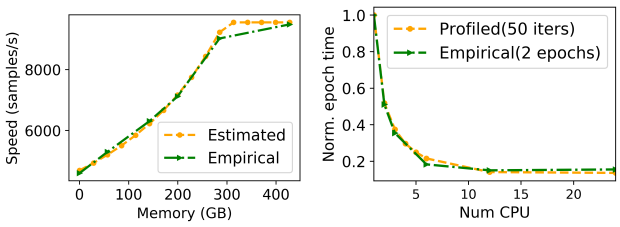
It is easy to see that naively profiling different combinations of CPU and memory can be very expensive. For instance, if the cost of profiling one combination of CPU, and memory for a job is 1 minute, then to profile all discrete combinations of CPU and memory (assuming allocation in units of 50GB) on a server with 24 CPUs and 500GB DRAM takes about 24*10 = 240 minutes (4 hours)!

To tackle this problem, Synergy introduces an optimistic profiling technique that exploits the predictability in the relationship between job throughput and memory allocation. We observe that, with DNN-specific, application-level caches like MinIO [39], it is easy to model the job throughput behaviour as we vary the amount of memory allocated to a job at fixed CPU allocation. This is because, MinIO ensures that a job gets a fixed number of cache hits per epoch. Synergy makes a conscious decision to use application-level MinIO cache instead of Page Cache because MinIO provides memory isolation across independent jobs sharing the machine. If we do not use MinIO, we will have to profile the model at discrete memory allocations which could result in increased profiling costs, and also potentially change the trends in profiling matrix. However, the use of MinIO in Synergy makes cache performance predictable and hence reduces Synergy 's profiling costs – allowing optimistic profiling.

For a given CPU allocation that determines the preprocessing speed, and a known storage bandwidth, it is easy to analytically model the job throughput for varying memory allocation. Therefore, we only need to empirically profile the job for varying CPU values at full memory allocation as shown in Figure 4. All the other entries can be estimated using the above technique. This leads to a 10× reduction in profiling time, bringing it down to 24 minutes! We experimentally validate this in Figure 5a. For a 8-GPU ResNet18 job, we compare the modeled job throughput using Synergy to the empirical results obtained by training the job for 2 epochs with varying memory allocations. As we see in Figure 5a, Synergy's estimations are within 3% of the empirical results, without having to actually run the model.

To further optimize profiling time, we observe that we do not require exact throughput values for a job with varying CPU allocations. We instead need a curve depicting the empirical job throughput. Therefore, instead of profiling the job for all possible CPU values, we pick discrete points for CPU profiling using the following algorithm. We start with the maximum CPU allocation and do a binary search on the CPU values to estimate job throughput. If the profiled point resulted in a throughput improvement that is less than a fixed threshold (say 10%), then we continue binary search on the lower half of CPU values, else we profile more points on the upper half. The idea here is to empirically profile CPU regions that show significant difference in job throughput, while skip those regions with little to no improvement in throughput. We experimentally show the efficacy of our CPU profiling technique in Fig 5b for a 1-GPU ResNet18 job. We compare the normalized job runtime (wrt 1 CPU) using empirical results averaged over 2 epochs of the job and Synergy's optimistic profiling averaged over 50 iterations (approximately, a minute per profile). Synergy is able to mimic the empirical job performance very closely, in under 8 minutes (using just 8 CPU profile points instead of 24). We believe that this is a reasonable overhead as it is incurred only once per lifetime of the job, which typically runs for hours.

After profiling a job on arrival, the job along with its resource sensitivity matrix is enqueued into the main scheduling queue, from which the scheduling policy picks a set of runnable jobs every round.



(a) Memory Validation (b) CPU validation

Figure 5: **Optimistic profiling**. The graphs compare the profiling results to empirical runs for ResNet18

3.2 Scheduling mechanism

Synergy performs round-based scheduling. At the beginning of each scheduling round, Synergy identifies a set of runnable jobs from the scheduling queue that can be packed on the cluster in the current round duration using a scheduling policy such as FIFO, SRTF, LAS, or FTF. Using the resource sensitivity matrix, Synergy packs these jobs onto the available servers in the cluster while satisfying the multi-dimensional resource constraints as opposed to simply performing a GPU-proportional allocation .

Job demand vector. To pack the jobs onto servers, we first construct a job demand vector that indicates the GPU demand, and best-case CPU and memory requirements for the job. We identify the best-case values using the resource sensitivity matrix. We pick the minimum value of CPU and memory that saturates the job throughput.

Packing a job with multi-dimensional resource demands is analogous to multi-dimensional bin packing problem which is NP hard [53]. Therefore, we first evaluate the efficacy of a naive greedy scheduling mechanism as an approximation to tackle the multi-dimensional resource allocation problem.

3.3 Synergy-GREEDY: Greedy Scheduling

A naive greedy multi-resource packing algorithm translates to a first-fit approximation of the multi-dimensional bin packing problem [20]. Given a job demand vector, the greedy algorithm picks the next runnable job decided by the scheduling policy, and places it on the server that can satisfy the job's demands in all dimensions. If no such server exists, the job is *skipped* over for this round and the next runnable job is checked for schedulability. Synergy-GREEDY thus introduces two major problems in the cluster -

- It can result in auxiliary resources being exhausted by jobs, while leaving GPUs underutilized, and fragmented. We show that GPU fragmentation in Synergy-GREEDY severely degrades cluster objectives (5.4).
- It also hurts the fairness of the scheduling policy as some jobs can be skipped over for a long time if their resource demands cannot be satisfied in the cluster.

The challenge ahead of us is to design a scheduling mechanism that eliminates GPU under-utilization due to fragmentation, and upholds the fairness properties of the given scheduling policy, while performing multi-dimensional resource allocation. Before we come up with a heuristic scheduling approach to tackle the above problems, one pertinent question is to understand how good is the allocation produced by our heuristic when compared to an optimal solution.

To this end, we first formulate a theoretical upper bound on the optimal throughput achieved by the cluster given a set of jobs and their resource sensitivity profiles. We then discuss the challenges associated with materializing the optimal allocation on a physical cluster and introduce Synergy-TUNE, an empirically close-to-optimal heuristic solution.

4 Scheduling Algorithms

We first present our formulation of an optimal allocation that provides an upper bound on the achievable cluster throughput.

4.1 Synergy-OPT

Our goal is to allocate CPU and memory to each job so as to maximize overall throughput, while guaranteeing that each job makes at least as much progress as it would do if we allocate its *GPU-proportional share*. It is not hard to show that our problem is NP-hard. So, we resort to finding approximate solutions using LP formulation. To find an *upperbound* on achievable throughput, we solve two LPs. In the interest of space, we describe the first LP formulation here, and summarize the challenges in operationalizing Synergy-OPT. A complete description of Synergy-OPT formulation and proof can be found in the extended version [38]. While the focus of this work is on homogeneous cluster, we show how our formulation can be extended to a heterogeneous GPU cluster in the extended version of the paper [38].

4.1.1 Finding ideal allocation

First, we assume an *idealized setting*: all the CPU and memory available across all the machines is present in one (super) machine. Say there are a total of s homogeneous machines in the cluster. We assume that, there is only one machine with G units of GPU, C units of CPU, and M units of memory. Note that, in reality G_i, C_i, and M_i denote the total GPU, CPU, and memory in each machine i, which is G/s, C/s, and M/s respectively in a homogeneous cluster. Based on this assumption, we find the ideal CPU (c^*_j) and memory (m^*_j) allocation for every job j (whose GPU demand is denoted by g_j) in the set of runnable jobs (J_t) for a round.

The variables of our LP are denoted by $y_{\{c,m,j\}}$, which should be interpreted as follows. If for a job $j \in J_t$, $y_{\{c,m,j\}} = 1$, then it means that in the LP solution c units of CPU and m units of memory are allocated. We further note that for

every job j, there is a variable $y_{\{c,m,j\}}$ for *for every possible* allocation of CPU and memory. We consider these variables in the discrete space as identified by our resource sensitivity matrix (W_j). $W_j[c,m]$ denotes the amount of progress made by job j per round if c units of CPU and m units of (RAM) memory are allocated to job j. For each machine $i \in [s]$, we denote C_g, M_g as the GPU-proportional allocation of CPU and memory. That is, $C_g = C_i/G_i * g_j$ and $M_g = M_i/G_i * g_j$. With a baseline GPU-proportional allocation strategy the progress a job makes in each round is equal to $W[Cg, Mg]$.

Our objective function is to maximize the throughput. We formulate it as follows using our LP variables.

$$\text{Maximize} \quad \sum_{j \in J_t} \sum_{[c,m]} W_j[c,m] \cdot y_{\{c,m,j\}} \tag{1}$$

Now, we enforce constraints such that LP solution is feasible in the idealized setting we talked about.

- Total CPU and memory allocated to jobs is no more than the total capacity available:

$$\sum_{j \in J_t} \sum_{[c,m]} c \cdot y_{\{c,m,j\}} \leq C \tag{2}$$

$$\sum_{j \in J_t} \sum_{[c,m]} m \cdot y_{\{c,m,j\}} \leq M \tag{3}$$

- We want the LP to allocate only one configuration of CPU and memory to each job.

$$\forall \text{j} \in J_t: \quad \sum_{[c,m]} y_{\{c,m,j\}} = 1 \tag{4}$$

- LP solution is atleast as good as the fair allocation.

$$\forall \text{j} \in J_t: \quad \sum_{[c,m]} W_j[c,m] \cdot y_{\{c,m,j\}} \geq W_j[C_g, M_g] \tag{5}$$

Theorem 4.1. Throughput achieved by LP(1-5) is at least the throughput achieved by an optimal solution to our problem.

Proof. Consider an optimal solution O to our problem. Suppose job j receives c^* units of CPU and m^* units of memory in O. Then we define the following feasible solution to our LP (1-5): Set $y_{c^*,m^*,j} = 1$. Clearly, this is a valid solution and satisfies constraints (1-4). □

In our experiments, we solve this as a Integer Linear Program (ILP) where $y_{\{c,m,j\}}$ takes boolean values. For every job, we define the total CPU (c^*_j) and memory (m^*_j) allocated by the optimal ILP solution as follows.

$$\text{For each job } j, \text{ define} \quad c^*_j := c \quad \text{if} \quad y_{\{c,m,j\}==1}. \tag{6}$$

$$\textit{and} \quad m^*_j := m \quad \text{if} \quad y_{\{c,m,j\}==1}. \tag{7}$$

4.1.2 Feasible Allocation on Multiple Machines

Recall that in the LP(1-5), we assumed that all the resources are present on a single machine. In reality, since these resources are spread across machines, we find a feasible allocation on multiple machines by solving a second LP. The objective here is to minimize the number of jobs that get fragmented to account for the communication overhead when jobs are split across machines. The variables of the second LP are denoted by $x_{i,j}$. Here index i denotes the machine and j denotes the job. If $x_{i,j} = 1$, it means that resources of job j (that g_j units of GPU, c^*_j units of CPU, and m^*_j units of memory) are allocated on machine i. Note that $x_{i,j}$ can be fractional; if so, then job j is split across multiple machines. We can prove that the solution to the second LP ensures that the total number of jobs that get fragmented is at most $3s$. Detailed formulation is in the extended version of the paper [38].

4.1.3 Challenges with operationalizing Synergy-OPT

While the allocations identified by Synergy-OPT provides an upper bound on the optimal cluster throughput, it is challenging to materialize these allocations in the real world due to two main reasons;

- Solving two LPs per scheduling round is computationally expensive. As cluster size and the number of jobs per round increases, the time to find an optimal allocation increases exponentially (§5.6)
- The allocation matrix obtained with the second LP can result in fractional GPU allocations when jobs are split across servers; for instance, a valid allocation might assign 3.3 GPUs on server 1 and 2.7 GPUs on server 2 for a 6 GPU job. Realizing such an allocation requires a heuristic rounding off strategy to ensure non-fractional GPU allocations, as GPU time or space sharing, and its impact on job performance is considered beyond the scope of this work.

4.2 Synergy-TUNE

We now describe Synergy-TUNE, our heuristic scheduling mechanism. Our goal is to design a scheduling mechanism that performs multi-dimensional resource allocation for DNN jobs, where the GPU demand is fixed, but the auxiliary resource allocations are fungible. In doing so, we want to ensure that (1) we do not affect the fairness properties of the scheduling policy used, (2) the expensive GPU resources are not underutilized.

Allocation Requirements. Synergy-TUNE's allocation must satisfy the following requirements.

- The GPU, CPU, and memory resources requested by a single-GPU job must all be allocated on the same server.
- A multi-GPU distributed-training job can either be consolidated on one machine, or split across multiple machines.

In the latter case, *the CPU and memory allocations must be proportional to GPU allocations across servers*. For instance, if a job requires (2GPU, 12 CPU, 300GB DRAM), then while splitting it across two servers, we need to ensure that each server gets (1GPU, 6CPU, 150GB DRAM). This is because, multi-GPU jobs train on a separate process on each GPU, and synchronize at regular intervals, i.e., after one or many iterations. The job performance will vary across processes if each GPU does not get the same ratio of resources, and will eventually proceed at the speed of the process with the lowest allocation of CPU and memory.

In a multi-tenant cluster, while carving out resources such as CPUs and memory for jobs, it is import to enforce fairness in terms of throughput achieved by individual jobs. We need to ensure that no job runs at a throughput lower than what it would have achieved if we allocated a GPU-proportional share of CPU and memory resources. Additionally, we need to respect the priority order of jobs identified by the scheduling policy. For instance, a FIFO scheduling policy can be implemented using a priority queue sorted by job arrival times. Synergy-TUNE identifies a set of runnable jobs for a round as the top n jobs from the scheduling queue, whose GPU demands can be exactly satisfied by the available servers in the cluster. Synergy-TUNE picks this runnable job set irrespective of the job's other resource demands - which are fungible. Note that, unlike Synergy-GREEDY, we do not skip over any jobs unless it cannot be scheduled (GPU demand cannot be met). Therefore, we never underutilize the GPUs when the cluster is at full load.

Next, Synergy-TUNE greedily packs each of these runnable jobs along multiple resource dimensions on one of the available servers, with the objective of minimizing fragmentation. To achieve this, Synergy-TUNE sorts the runnable jobs by their GPU demands, followed by CPU, and memory demand. For each job j in order, Synergy-TUNE then picks the server with the least amount of free resources just enough to fit the demand vector of j. If it is a multi-GPU job, then we find a minimum set of servers with sufficient GPU availability that can fit the job's demands in entirety. However, it is possible that the job cannot fit in the cluster along all dimensions. In such a case,

1. We check if the job's demand vector is greater than proportional share of resources, In this case, we switch the job's demand to GPU-proportional share and retry.

2. If the job still does not fit the cluster, or if the job's demand vector was less than or equal to GPU proportional allocation in step (1), then, we do the following.

(a) We repeat step (1) ignoring the job's CPU and memory requirements. We find a server that can just satisfy the job's GPU requirements. We know by construction that there is atleast one job on this server, which is allocated more than GPU-proportional share of resources. We identify the job or a set of jobs (J_s) on this server by switching whom to GPU-proportional share, we can release just as much resources required by the current job j. We switch the jobs in J_s to fair-share and by design, job j will fit this server.

(b) We continue this recursively for all runnable jobs.

In the worst case, all the running jobs in a round could be allocated GPU-proportional share of resources. Therefore, Synergy ensures that its allocations results in job throughputs that are never worse than GPU-proportional allocation. In §5.6, we empirically compare Synergy-TUNE to Synergy-OPT showing that it is practical and near-optimal.

4.3 Implementation

We implement Synergy and an associated simulator in Python. Our scheduler is event-driven. There is a global event queue where job arrivals, schedule events, and deploy events are queued. These events are handled in the order of their arrival time. There is a priority job queue, where all the jobs arriving into the cluster are added, post profiling. This queue is sorted by the priorty metric decided by the scheduling policy; for instance, SRTF sorts the jobs in the order of job remaining time.

When a schedule event occurs, the scheduler collects a list of runnable jobs from the job queue and identifies the appropriate placement for these jobs for the following round, either using Synergy-GREEDY, Synergy-TUNE or Synergy-OPT. Then when a deploy event is triggered, these allocations are deployed on to the cluster. By default, every job requests for a lease update to continue running on the same server [42]. The scheduler then either grants a lease update or terminates the lease for the job, adding it back to the job queue.

The scheduler and the DNN jobs interact via a thin API provided by the Synergy data iterator. DNN job scripts must be updated to call the Synergy iterator which is a wrapper around the default PyTorch [8] and DALI [7] iterators. The iterator handles registering the job with the scheduler, and appropriately sending lease updates. It also checkpoints the job to a shared storage if its lease is terminated. The iterator also synchronizes across GPU processes for a multi-GPU job to ensure that each process makes identical progress. We use `gRPC` [4] to communicate between the scheduler and the jobs.

We implement Synergy-OPT in `cvxpy` [19] for use in our simulator. The optimistic profiling module is also implemented in Python, and it profiles the incoming jobs hooked to the Synergy iterator, prior to the job's initial addition to the scheduling queue (a one time overhead for each job).

5 Evaluation

In this section, we use trace-driven simulations from production cluster traces, and physical cluster deployment to evaluate

Task	Model	Dataset
Image	Shufflenetv2 [58] AlexNet [32] Resnet18 [28] MobileNetv2 [50] ResNet50 [28]	ImageNet [49]
Language	GNMT [54] LSTM [47] Transformer-XL [16]	WMT16 [9] Wikitext-2 [36] Wikitext-103 [36]
Speech	M5 [15] DeepSpeech [27]	Free Music [17] LibriSpeech [45]

Table 4: **Models used in this work**.

Policy (Metric)	Workload Split	Mechanism	Time (hrs) Deploy	Time (hrs) Simulate
FIFO (Makespan)	60-30-10	Proportional	16	15.67
		Tune	11.6	11.33
		Opt	-	11.01
SRTF (Avg JCT)	30-60-10	Proportional	4.81	4.52
		Tune	3.21	3.19
		Opt	-	3.06
SRTF (99 Percentile JCT)	30-60-10	Proportional	17.32	16.85
		Tune	8.59	8.54
		Opt	-	8.21

Table 5: **Physical cluster experiments**. This table compares the makespan, average JCT, and 99th percentile JCT for two different traces; (1) a static trace using FIFO (2) a dynamic trace using SRTF. Synergy-TUNE improves makespan by 1.4×, average JCT by 1.5 × and 99th percentile JCT by 2×.

the efficacy of Synergy. Our evaluation seeks to answer the following questions.

- Does Synergy's resource-sensitive scheduling improve cluster objectives such as makespan and average JCT in a physical cluster (§5.2) and in trace-driven simulations of large-scale clusters (§5.3) ?
- How does Synergy-TUNE and Synergy-GREEDY perform with different workload splits and how well do they utilize available resources (§5.4)?
- How does Synergy perform on different CPU:GPU ratios (§5.5)?
- Compare Synergy-TUNE to Synergy-OPT (§5.6)?
- Compare Synergy to big data schedulers (§5.7)?

5.1 Experimental setup

Clusters. Our experiments run on both a physical and a large simulated, homogeneous cluster. Our experiments are performed on state-of-the-art internal servers at Microsoft - these servers are part of a larger multi-tenant cluster. We run physical cluster experiments on a cluster with 32 V100 GPUs across 4 servers. Each server has 500GB DRAM, 24 CPU cores, and 8 GPUs. Unless otherwise specified, our experiments assume a CPU:GPU ratio of 3 and fair-share memory allocation of 62.5GB per GPU, matching the server configurations above. For simulations, we assume two cluster sizes; a 128 GPU cluster across 16 servers and a 512 GPU cluster across 64 machines, where each machine resembles the physical server configuration mentioned above.

Models. Our experiments consider 10 different DNNs (CNNs, RNNs, and LSTMs) as shown in Table 4. We categorize these models by task (image, language, and speech) and assign a certain weight to these tasks in our traces. We call this a workload *split*. For instance, if the split for a given trace is (30,40,30), then the percentage of image, language, and speech models in the job trace is 30%, 40% and 30% respectively. All experiments are performed on PyTorch 1.1.0.

Traces. We run our physical and simulated experiments using publicly available production traces from Microsoft Philly cluster [5].We show evaluation with the actual Philly trace preserving the job GPU demand, arrival time, and duration, on a cluster of 512 GPUs in §5.3.1. We use a subrange of the trace containing 8000 jobs.

However, to comprehensively evaluate how Synergy reacts to varying cluster load, workload composition, and job duration, for all other experiments, we construct a production-derived trace as follows: we extract job GPU demand from the Philly trace and assign a model based on the chosen *split*. We then appropriately scale the job runtime and arrival time for the chosen cluster size, while keeping the job duration distribution similar to the one in Philly trace as follows:

- **Duration**. The duration of each job for the baseline GPU-proportional allocation is sampled from an exponential distribution: the job duration is set to 10^x minutes, where x is drawn uniformly from [1.5,3] with 80% probability, and from [3,4] with 20% probability similar to the trace duration used in prior work [42].
- **Arrival**. We classify derived traces into two kinds based on the job arrival time : (1) a *static* trace where all the jobs arrive at the start of the workload, and (2) a *dynamic* trace, where the job arrival time is determined by load, a Poisson distribution at a rate λ.

The derived traces with varying job arrival rates uses a 128 GPU cluster. In both cases, we report the average metrics such as JCT across a set of 1000 jobs in steady state.

For the physical cluster experiment, we choose a fixed arrival rate for the derived trace that keeps our cluster at *full load* (GPU demand of all runnable jobs > available GPUs in the cluster). For the simulated experiments, we vary the load λ on the cluster to evaluate its impact on cluster metrics. For the simulated experiments, we show results for two trace categories - (1) all jobs request single-GPU (2) multi-GPU distributed training jobs that request upto 16 GPUs.

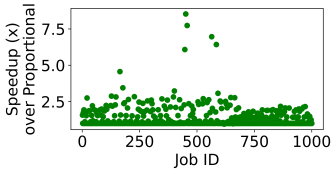
Policies and metrics. We evaluate Synergy against GPU-proportional scheduling for 4 different scheduling polices; FIFO, SRTF, LAS, and FTF. For a static trace, we measure makespan (time to complete all jobs submitted at the beginning of the trace) and for the dynamic job traces, we measure

Policy	Avg JCT(hrs) SRTF	LAS	FIFO
GPU-prop.	30	32	71
Synergy	26	28	62

(a) Average JCT with Synergy

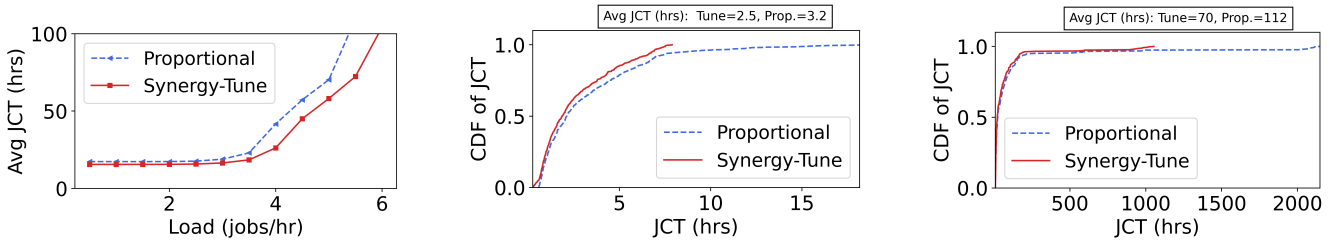
JCT (hrs)		Short	Long
Avg	Prop.	2	80
	Synergy	1.7	68
99p	Prop.	9	660
	Synergy	4	641

(b) Cluster metrics (SRTF)



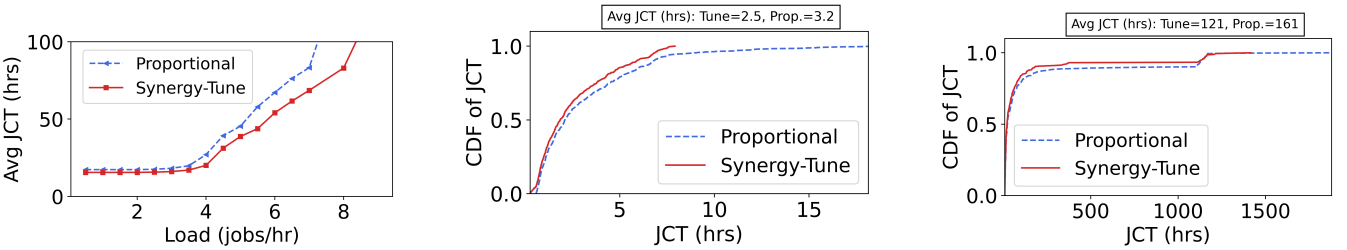
(c) JCT speedup across jobs

Figure 6: **Evaluation on Philly Trace**. On a real production trace, Synergy improves avg JCT across a range of scheduling policies over GPU-proportional scheduling. The JCT of individual jobs improves by upto 9× with Synergy.



(a) LAS (multi) (b) CDF of JCT at load 4 (short) (c) CDF of JCT at load 4 (long)

Figure 7: **Average JCT and CDF of long and short jobs for LAS policy**.



(a) SRTF (multi) (b) CDF of JCT at load 5.5 (short) (c) CDF of JCT at load 5.5 (long)

Figure 8: **Average JCT and CDF of long and short jobs for SRTF policy**.

the average job completion time (JCT) of a subset of jobs in steady state (cluster at full load), and their CDF.

5.2 End-to-End Physical Cluster Experiments

For the physical cluster experiments, we run a Synergy-TUNE (*tune*) and GPU-proportional allocation (*proportional*) for two different workload traces. (1) A static production-derived trace of 100 jobs with a *split* (60,30,10), scheduled using FIFO and evaluated for makespan. (2) A dynamic production-derived trace with continuous job arrivals and a split of (30,60,10), scheduled using SRTF and evaluated for average and 99th percentile JCT. Both scenarios use an appropriately sized trace that keeps the cluster fully loaded. We compare the obtained results to that of the simulator by replaying the same trace. Additionally, we compare our metrics to the upper bound generated by the optimal solution, Synergy-OPT (*opt*). The results are shown in Table 5.

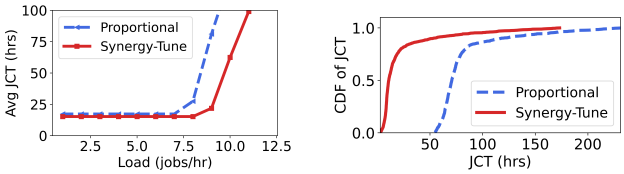
Synergy-TUNE reduces the makespan of static trace by 1.4× when compared to GPU-proportional allocation. For the dynamic trace, Synergy-TUNE reduces average JCT of steady-state jobs by 1.5× while reducing the 99th percentile JCT of these jobs by 2× as shown in Table 5.

We compare the observed results from physical experiments to the same trace replayed on our simulator. As shown in Table 5, the difference between metrics in real and simulated clusters are less than 5%, demonstrating the fidelity of the simulator. We also see from Table 5 that the cluster objectives achieved by Synergy-TUNE are within 4% of the optimal solution in this case. We do not deploy the optimal allocations due to the challenges enumerated in §4.1.3

5.3 End-to-end results in simulation

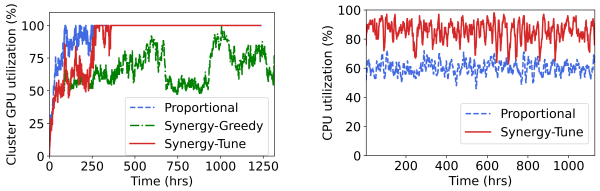
5.3.1 Simulation with production traces

We run simulated experiments on a cluster of 512 GPUs across 64 servers using a subrange of the publicly available Philly trace published by Microsoft [5]. We assume a workload split of (20,70,10) for this trial. Table 6a lists the average JCT with Synergy and GPU-proportional scheduling for three differ-



(a) FIFO (single) (b) CDF (9 jobs/hr)

Figure 9: **Average JCT and CDF for FIFO**. Synergy improves the average JCT significantly compared to alllocation for varying cluster load. At a load of 9 jobs/hr, Synergy reduces average JCT from 81hrs to 22hrs, which is close to the upperbound of 20hrs predicted by Synergy-OPT.



(a) GPU utilization (b) CPU utilization

Figure 10: Cluster resource utilization

ent scheduling policies. Across all policies, Synergy is able to reduce the average JCT compared to GPU-proportional scheduling due to better split of resources between jobs. The gains in Synergy can be attributed to reallocating the underutilized resources from a job to a different, resource-sensitive job whose throughput can improve with the increased allocation.

We show a detailed overview of the average and 99th percentile JCT for SRTF policy in Table 6b.We split the set of 1000 monitored jobs into short (JCT $<$ 4 hrs) and long jobs. Synergy reduces the tail of the distribution by 2.2$\times$ for short jobs and the average JCT of both long and short jobs by 15%. For each of the 1000 monitored jobs, we plot the individual job speedup with respect to GPU-proportional scheduling in Figure 6c. We see that Synergy speeds up jobs by upto 9$\times$ using better resource allocations.

5.3.2 Simulation with varying load

We run simulated experiments on a cluster of 128 GPUs across 16 servers using production-derived traces. We evaluate Synergy against GPU-proportional allocation mechanism for 4 different scheduling policies - FIFO, SRTF, LAS and FTF. We run dynamic workload traces, where jobs arrive continuously at a rate governed by a Poisson distribution. We show results for both single-GPU traces (where all jobs request 1 GPU) and multi-GPU traces (where jobs request upto 16 GPUs). Our metric of evaluation is the average JCT of a set of 1000 jobs in cluster steady state.

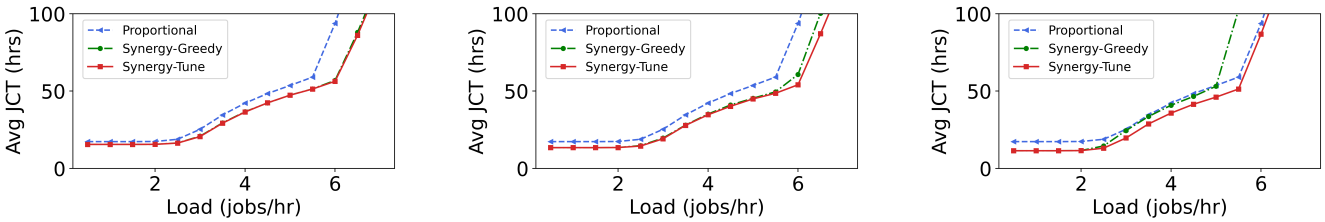
We show the results for three scenarios : LAS (multi-GPU trace) in Figure 7, SRTF (multi-GPU trace) in Figure 8, and FIFO (single GPU trace) in Figure 9. In all cases, we assume a workload split of (20,70,10). We plot both average JCT and the CDF of job completion times for a specific cluster load in both scenarios. For the multi-GPU trace, we split the CDF into those for short and long jobs to distinctly differentiate the tail of the distribution. We make three key observations.

First, Synergy-TUNE improves average JCT by up to 3.4$\times$ in the single-GPU trace, and up to 1.6$\times$ in the multi-GPU trace by speeding up resource sensitive jobs with disproportionate allocation. The improvement in average JCT is higher as the load increases, because at low load the cluster is not at full capacity. As load increases, jobs start to get queued and incur queuing delay before being scheduled on the cluster. Since Synergy significantly speeds up individual jobs using disproportionate resource allocation, pending jobs can get scheduled faster, thereby reducing their queuing delays. Therefore Synergy improves cluster metrics by both reducing queuing delays and speeding up individual jobs. Note that, in GPU-proportional allocation, at higher loads, all CPUs and memory in the system are allocated to the running jobs but they can still be underutilized by individual jobs. We show later in Figure 10b, how Synergy's resource-sensitivity aware allocation improves CPU utilization in the system compared to GPU-proportional allocation. At low load, jobs are spread across the cluster and the unallocated CPU and memory is assigned to the jobs that benefit from additional auxiliary resources. Second, Synergy-TUNE is able to sustain a larger cluster load than GPU-proportional allocation. For multi-GPU scheduling with LAS, Synergy-TUNE reduced the 95th percentile JCT of long jobs by 2$\times$. Third, the average JCT achieved with Synergy-TUNE is within 10% of the optimal solution in all cases.

Similarly, for FTF scheduling policy, Synergy-TUNE observed 2.3$\times$ and 2$\times$ improvement in average JCT for a single-GPU and multi-GPU trace respectively.

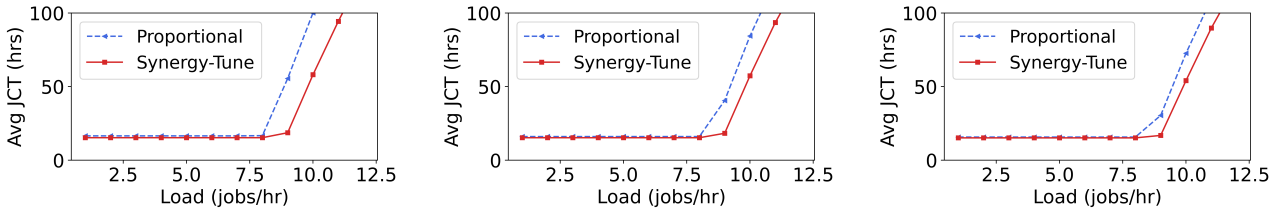
5.4 Impact of workload split

Workload split decides the percentage of resource sensitive jobs in the workload. As the percentage of speech and image models increase in the trace, there may not be enough spare CPU and memory resources to perform disproportionate allocation, as they are mostly CPU- and memory-hungry. Figure 11 plots the average JCT with varying load for 3 different workload splits with FIFO scheduling for multi-GPU jobs. As the percentage of resource-sensitive jobs increase, we observe that Synergy-GREEDY breaks down, and ends up degrading JCTs significantly compared to a GPU-proportional allocation. This is because, the naive greedy technique results in resource fragmentation when the demand along CPU and memory dimensions are high, leaving several GPUs underutilized. Whereas, by the design of Synergy-TUNE, it allocates at least as many resources required to achieve the throughput of GPU-proportional allocation; therefore, even in the worst case workload split shown in Figure 11c, where all the jobs



(a) Split=(20,70,10) (b) Split=(33,33,33) (c) Split=(50,0,50)

Figure 11: Evaluation of Synergy with varying workload split



(a) Ratio 4 (b) Ratio 5 (c) Ratio 6

Figure 12: Evaluation of Synergy across different CPU:GPU Ratio

are CPU- and memory-sensitive, Synergy-TUNE performs as good as GPU-proportional allocation.

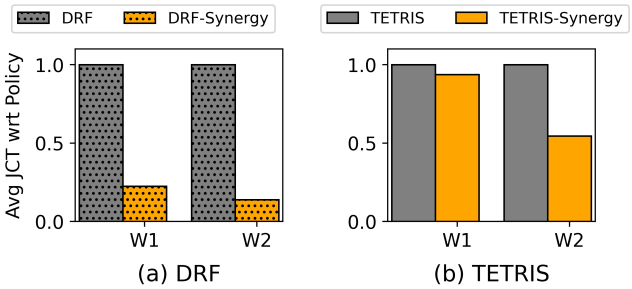
Resource utilization. Figure 10a plots the GPU allocation over time for the workload in Figure 11c at a load of 5.5 jobs/hr where the cluster GPU demand is higher than 100%. While Synergy-TUNE is able to sustain a higher load by finishing jobs faster, Synergy-GREEDY severely under-utilizes GPU resources throughout the workload, trading it off for higher CPU and memory allocation. At low loads, as shown in Figure 10b, GPU-proportional allocation only utilized 60% of the available CPU resources, while Synergy-TUNE utilized it up to a 90%, resulting in 1.5× lower average JCT.

5.5 Impact of CPU:GPU ratio

While our prior experiments assume a CPU:GPU ratio of 3 (similar to the NVIDIA DGX-2), Figure 12 plots the average JCT for a FIFO scheduler on a single-GPU trace as we increase cluster load and vary the CPU:GPU ratio from 4 to 6 (corresponding to other server SKUs in Table 2b). As the CPU:GPU ratio in a server increases, the baseline GPU-proportional scheduler gets more CPU cores per GPU, thereby reducing data stalls in the baseline. This in turn, reduces the gap between GPU-proportional and Synergy-TUNE. Despite that, at a load of 9 jobs/hr, Synergy-TUNE lowers the avg JCT by 3.4×, 3×, 2.2×, and 1.8× for a CPU:GPU ratio for 3, 4, 5 and 6 respectively.

5.6 Comparison to Synergy-OPT

Calculating optimal allocations for every scheduling round with Synergy-OPT can be quite expensive, especially for large



(a) DRF (b) TETRIS

Figure 13: Comparison to big data scheduling policies

cluster sizes. We experimentally validated that the time taken for per-round allocations for Synergy-OPT increases exponentially with increasing cluster sizes, while that for Synergy-TUNE is hardly a second. We also show experimentally that the allocations given by Synergy-TUNE are close to those estimated by Synergy-OPT in §5.2 and §5.3.2. For a cluster size of 128 GPUs used in our experiments, Synergy-TUNE converges at allocations that are within 10% of the optimal value, 200× faster than Synergy-OPT.

5.7 Comparison to DRF and Tetris

Big data schedulers like Dominant Resource Fairness (DRF) [21] and Tetris [23] have explored multi-dimensional resource allocation for map-reduce jobs. DNN jobs have different properties when compared to big-data jobs. DNN jobs are gang-scheduled, meaning they can run only when all the GPUs requested by them are available on the cluster at once. Further, the auxiliary resource requirements like CPU and memory are fungible unlike the GPU demand. DRF and Tetris assume resources to be statically allocated throughout the life-

time of a job, whereas Synergy assumes these resources to be fungible and could result in varied allocations throughout the lifetime of a DNN job. Furthermore, profiling the DNN job's resource demands is unique to Synergy; big data schedulers assume that the job request already encodes resource demands across all dimensions. To evaluate Synergy against these policies, we assume that the best-case resource requirement for CPU and memory is fed as input to the bigdata scheduling policies using Synergy's profiling mechanism.

On a cluster of 128 GPUs, we evaluate these policies on two different workload compositions : W1 (20,70,10), and W2 (50,0,50) and compare the naive policy with its Synergy-variant, which allows resource tuning. W1 represents a workload split with a good mix of resource-sensitive as well as resource-insensitive jobs. W2 is a workload dominated by resource-sensitive jobs, which is one of the worst-case scenarios for multi-dimensional scheduling as it could lead to GPU fragmentation (explained in §5.4)

We plot the results in Figure 13. Tuning resource allocation across jobs using Synergy reduced the average JCT of DRF by 7.2× and that of Tetris by 1.8× for the workload split W2. This is because Synergy is able to allocate auxiliary resources in a fungible-manner every round, whereas the big-data scheduler's static allocations performs similar to greedy techniques, resulting in GPU fragmentation, and thereby degrading the overall cluster metrics. Synergy performs the best in each scenario as it uses the best-case resource demands of jobs to perform fungible, disproportionate allocation.

6 Discussion and Future Work

In this section, we elaborate on some of the assumptions made by Synergy, derived from our experiences with large scale deployed cluster schedulers at Microsoft, and discuss what happens if these assumptions are relaxed.

Homogeneous clusters. Scheduling in Synergy assumes that the GPU cluster is homogeneous. This assumption is based on the practical observation that our clusters have thousands of accelerators per homogeneous cluster [5]. While there is heterogeneity in hardware across clusters, it is often the case that users select one homogeneous cluster to run their job in production. For instance, a production cluster could have two homogeneous virtual clusters (VCs), each comprising of a specific generation of GPU. Each VC is managed separately, and assigned to a specific task - training or inference, for predictable performance. While recent works have explored the impact of blurring these boundaries and scheduling across heterogeneous hardware [11, 33, 42], such co-scheduling poses several practical challenges [52]. For example, some tasks such as low-latency inference are business-critical, user-facing applications which need to run on specific hardware, and need data isolation. Others have specific GPU memory requirements, or need advanced hardware features like NVLink. Hence, users in our production settings specify a specific instance type to run each of their jobs on. Hence it is useful for a scheduler to optimize resource utilization in the context of homogeneous clusters. That said, Synergy 's ideas can also be extended to a heterogeneous cluster by profiling CPU and memory requirements along an additional dimension - GPU type, at an additional profiling cost. The optimal algorithm can then maximize throughput based on a 3-dimensional resource-sensitivity matrix W_j. We present the formulation for this in the extended version of the paper [38].

Use of MinIO. Synergy assumes the use of MinIO [39] because it is a DNN-aware caching mechanism that outperforms traditional OS page caching and allows performance predictability. It provides resource isolation and reduces storage fetch stalls [39]. If we do not use MinIO, we will have to profile the model at discrete memory allocations which will increase the profiling costs, and also potentially change the trends in profiling matrix.

Preprocessing overhead. Preprocessing for vision tasks includes random cropping and transformations of the image in the critical path. Reusing the same transformed images across epochs hurts accuracy [34, 37, 39], whereas it is practically infeasible to pre-process offline due to the prohibitive storage cost (dataset size * epochs). It is possible to alter the extent of CPU intensiveness by varying the number of augmentations performed. In this work, we have assumed that the augmentations required for each model are as specified by the published models themselves and we do not change this so as to not affect accuracy. On the horizon, we do observe recent schemes such as RandAugment [14], AutoAugment [13] which consider more computationally-intensive augmentation schemes (and associated accuracy gains). Such a rising trend in extreme preprocessing, makes a strong case for a system like Synergy.

Sharing storage and network. In our paper, we show how to reallocate CPUs and memory across jobs resident on the same server, for example, by co-locating a CPU-intensive task with a non CPU-intensive task. For our DNN training jobs, we assume that a dataset is downloaded locally and loaded into server memory when the job is started (constrained by the memory allocation limits). Prior work has similarly looked at co-locating network-intensive jobs with non network-intensive jobs [26, 35], but unlike Synergy, reallocation of shared network bandwidth is not explicitly handled by those schedulers. We leave it to future work to explore how ideas in Synergy can also be extended to reason about demands that individual jobs place on storage and network bandwidths.

GPU elasticity and sharing. While some recent works explore transparently changing the GPU allocation during the life of a job [48], the impact of changing batch sizes and hyperparameters on training accuracy is unclear for a wide variety of tasks. It is therefore practical to assume that the

GPU demand of a job is constant throughout its lifetime as is the case for jobs in our production clusters.

Synergy works by improving the throughput of jobs that are bottlenecked on data stalls. For jobs that have data stalls, GPU efficiency cannot be improved by multiplexing (spatial sharing) because they are waiting for input data. However, for a subset of jobs that are insensitive to auxiliary resource allocation, GPUs could be multiplexed between jobs. It would be interesting to explore how to impart resource-sensitivity awareness alongside GPU spatial sharing, which we leave for future work.

Tradeoff between consolidation and allocation. When multi-GPU jobs are split across physical servers, they may incur a penalty due to network communication [41, 55]. DNN jobs therefore prefer consolidation. In this work, we assume that no more than a server's worth of CPU or memory resources can be allocated to a job if its GPU demands can be satisfied by one server. However, we find that some jobs may benefit from giving up consolidation if the throughput gain due to increased CPU or memory allocation is higher than the penalty due to splitting. We leave the exploration of the trade off between consolidation and allocation, while taking into account the network overhead, to future work.

Leveraging model and pipeline parallelism. Our evaluation assumes distributed data-parallel jobs. But model and pipeline parallel execution schemes also have an input stage that ingest and pre-process data. Unlike data-parallel training, each stage in the pipeline might have a different CPU-GPU and memory-GPU requirement. While these jobs would have to be profiled to identify the CPU and memory sensitivity of each stage of the pipeline, Synergy 's contributions directly carry forward to such settings.

7 Related Work

DNN cluster schedulers. A number of recent schedulers for DNN workloads each focus on improving a certain objective; Cluster utilization (Gandiva [55]), JCT (Tiresias [26]), and fairness (Themis [35], Gandiva-Fair [11]). Some have also looked at exploiting performance heterogeneity among accelerators to improve cluster objectives [33, 42]. All these schedulers assume GPU to be the dominant resource in the scheduling task; i.e., a user requests a fixed number of GPUs for her DNN job, and when the requested number of GPUs are all available, the job is scheduled to run. Rather than allocating a fixed number of GPUs, building on GPU-elasticity for a single job [44], some recent schedulers like AFS [30] and Pollux [48] leverage throughput metrics to provide GPU elasticity in multi-tenant clusters (in addition to tuning batch size and learning rate). However, in all these cases, auxiliary resources such as CPU and memory are allocated proportional to the number of GPUs allocated to the job. Existing schedulers thus ignore *resource-sensitivity* of the DNN tasks to CPU and memory. Synergy shows that, irrespective of the number of GPUs allocated, auxiliary resource-sensitive allocation is crucial to achieve better cluster utilization.

Big data schedulers. Our work builds upon the insights drawn from the rich literature of schedulers for big data jobs [21, 23–25, 29, 51]. Big data schedulers like Tetris [23], and DRF [21] have looked at the problem of multi dimensional resource allocation for big data jobs. They propose new scheduling policies aimed at optimizing a specific cluster objective for jobs whose resource demands are prior known. In contrast, the primary resource in a DNN job is the accelerator (GPU), whose requirement is specified by the job; other resources are fungible. Our work exploits this insight to perform disproportionate allocations by profiling job resource sensitivity, and then appropriately packing them onto servers.

Data stalls. Recent, deep characterization studies explored the impact of CPU and memory on individual DNN jobs [39, 40] Unlike prior work that focuses on individual jobs, the focus of our paper is on the tricks we can play when we schedule multiple jobs together in a cluster.

Disaggregated data prep. There have been recent orthogonal efforts that aim at reducing the cost of data preprocessing, and thereby the load on CPUs using disaggregated data prep [59]. However, one has to pay the network cost of shuffling preprocessed tensors, which could quickly become the bottleneck especially for vision models with rich datasets. Synergy on the other hand, assumes standard pre-processing pipelines at the training servers, and aims to reduce the cost of pre-processing using better resource allocation.

8 Conclusion

This paper introduces Synergy, a resource-sensitive scheduler for DNN training jobs. Synergy is based on the insight that not all jobs exhibit the same level of sensitivity to CPU and memory allocation during DNN training; breaking the shackles of GPU-proportional allocation can result in improved utilization of existing cluster resources and improved job and cluster-wide objectives. Our experiments on physical and large simulated clusters show that Synergy can reduce average JCT by upto 3.4× over GPU-proportional allocation.

Acknowledgements

We thank our shepherd Ravi Netravali, the anonymous OSDI reviewers, members of the UT SaSLab, and many of our MSR colleagues for their invaluable feedback that made this work better. Sincere thanks to the fellow Project Fiddle interns Kshiteej Mahajan and Andrew Or for their contributions to the simulator infrastructure. We thank Microsoft Research for their generous support of JM's internships, and for the many resources required to develop and evaluate this work.

References

[1] Amazon EC2 P3 - Ideal for Machine Learning and HPC - AWS. https://aws.amazon.com/ec2/instance-types/#p3.

[2] Azure NC_v3 Series. https://docs.microsoft.com/en-us/azure/virtual-machines/ncv3-series.

[3] Azure ND_v2 Series. https://docs.microsoft.com/en-us/azure/virtual-machines/ndv2-series.

[4] gRPC. https://grpc.io/.

[5] Microsoft philly traces. https://github.com/msr-fiddle/philly-traces.

[6] NVIDIA DGX-2: Enterprise AI Research System. https://www.nvidia.com/en-us/data-center/dgx-2/.

[7] NVIDIA DALI. https://github.com/NVIDIA/DALI, 2018.

[8] Pytorch. https://github.com/pytorch/pytorch, 2019.

[9] Wmt16. http://www.statmt.org/wmt16/, 2020.

[10] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[11] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 1:1–1:16. ACM, 2020.

[12] Mark Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection scheduling in web servers. In *2nd USENIX Symposium on Internet Technologies and Systems, USITS'99, Boulder, Colorado, USA, October 11-14, 1999*. USENIX, 1999.

[13] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation strategies from data. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 113–123. Computer Vision Foundation / IEEE, 2019.

[14] Ekin Dogus Cubuk, Barret Zoph, Jon Shlens, and Quoc Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[15] Wei Dai, Chia Dai, Shuhui Qu, Juncheng Li, and Samarjit Das. Very deep convolutional neural networks for raw waveforms. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 421–425. IEEE, 2017.

[16] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2978–2988. Association for Computational Linguistics, 2019.

[17] Michaël Defferrard, Kirell Benzi, Pierre Vandergheynst, and Xavier Bresson. Fma: A dataset for music analysis. *arXiv preprint arXiv:1612.01840*, 2016.

[18] NVIDIA DGX-1. https://www.nvidia.com/en-us/data-center/dgx-1/.

[19] Steven Diamond and Stephen P. Boyd. CVXPY: A python-embedded modeling language for convex optimization. *J. Mach. Learn. Res.*, 17:83:1–83:5, 2016.

[20] György Dósa and Jirí Sgall. First fit bin packing: A tight analysis. In Natacha Portier and Thomas Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, volume 20 of *LIPIcs*, pages 538–549. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[22] Google. Open images dataset. https://opensource.google/projects/open-images-dataset.

[23] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 455–466. ACM, 2014.

[24] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 65–80. USENIX Association, 2016.

[25] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, Savannah, GA, November 2016. USENIX Association.

[26] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 485–500. USENIX Association, 2019.

[27] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[29] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[30] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 721–739. USENIX Association, 2021.

[31] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[33] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 31:1–31:16. ACM, 2020.

[34] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyung-Geun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 537–550. USENIX Association, 2021.

[35] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 289–304. USENIX Association, 2020.

[36] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[37] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Checkfreq: Frequent, fine-grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 203–216. USENIX Association, 2021.

[38] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Synergy: Resource sensitive DNN scheduling in multi-tenant clusters. *CoRR*, abs/2110.06073, 2021.

[39] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *Proc. VLDB Endow.*, 14(5):771–784, 2021.

[40] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, 2021.

[41] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15. ACM, 2019.

[42] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 481–498. USENIX Association, 2020.

[43] Misja Nuyens and Adam Wierman. The foreground-background queue: A survey. *Perform. Evaluation*, 65(3-4):286–307, 2008.

[44] Andrew Or, Haoyu Zhang, and Michael J. Freedman. Resource elasticity in distributed deep learning. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[45] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015, South Brisbane, Queensland, Australia, April 19-24, 2015*, pages 5206–5210. IEEE, 2015.

[46] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 3:1–3:14. ACM, 2018.

[47] PyTorch. Word-level language modeling rnn. https://github.com/pytorch/examples/tree/master/word_language_model.

[48] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 2021.

[49] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[50] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[51] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 5:1–5:16. ACM, 2013.

[52] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, 2022.

[53] Gerhard J. Woeginger. There is no asymptotic PTAS for two-dimensional vector packing. *Inf. Process. Lett.*, 64(6):293–297, 1997.

[54] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[55] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[56] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 533–548. USENIX Association, 2020.

[57] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10,*

Boston, MA, USA, June 22, 2010. USENIX Association, 2010.

[58] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[59] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding and co-designing the data ingestion pipeline for industry-scale recsys training. *CoRR*, abs/2108.09373, 2021.

CAP-VMs: Capability-Based Isolation and Sharing in the Cloud

Vasily A. Sartakov
Imperial College London

Lluís Vilanova
Imperial College London

David Eyers
University of Otago

Takahiro Shinagawa
The University of Tokyo

Peter Pietzuch
Imperial College London

Abstract

Cloud stacks must isolate application components, while permitting efficient data sharing between components deployed on the same physical host. Traditionally, the MMU enforces isolation and permits sharing at page granularity. MMU approaches, however, lead to cloud stacks with large TCBs in kernel space, and page granularity requires inefficient OS interfaces for data sharing. Forthcoming CPUs with hardware support for *memory capabilities* offer new opportunities to implement isolation and sharing at a finer granularity.

We describe *cVMs*, a new VM-like abstraction that uses memory capabilities to isolate application components while supporting efficient data sharing, all without mandating application code to be capability-aware. cVMs share a single virtual address space safely, each having only capabilities to access its own memory. A cVM may include a library OS, thus minimizing its dependency on the cloud environment. cVMs efficiently exchange data through two capability-based primitives assisted by a small trusted monitor: (i) an asynchronous read/write interface to buffers shared between cVMs; and (ii) a call interface to transfer control between cVMs. Using these two primitives, we build more expressive mechanisms for efficient cross-cVM communication. Our prototype implementation using CHERI RISC-V capabilities shows that cVMs isolate services (Redis and Python) with low overhead while improving data sharing.

1 Introduction

Cloud environments require application compartmentalization. Today, isolation between application components is enforced by virtual machines (VMs) [10, 32, 63] and containers [2, 40], either separately or in combination. Yet, current applications push the limits of these mechanisms in terms of performance and security: when application components communicate heavily with each other, VMs and containers add substantial overheads, even when they are co-located to improve communication performance; furthermore, the implementation of the isolation mechanisms may also rely on a large trusted computing base (TCB).

VMs provide strong isolation through a relatively narrow hardware interface. Since a guest VM has its own OS kernel, its TCB can be reduced to a relatively small hypervisor, which multiplexes VM access to the hardware [56]. Efficient inter-VM data sharing, however, is challenging to achieve due to performance and page granularity trade-offs [17, 71].

In contrast, *containers* isolate processes into groups [2] and provide faster inter-process communication (IPC) primitives, including pipes, shared memory, and sockets. Similar to VMs, they face problems of page-level sharing granularity and overheads due to frequent user/kernel transitions. Their richer IPC primitives for data sharing come at the cost of a larger TCB—a shared OS kernel implements both namespace isolation between process groups and complex IPC primitives, increasing the likelihood of security vulnerabilities.

Existing cloud stacks thus face a fundamental tension when application components are compartmentalized but must communicate. They must either copy data or modify page tables, both of which are expensive operations that involve a privileged intermediary, e.g., a hypervisor or OS kernel, and lead to coarse-grained interfaces designed around page granularity.

In this work, we explore a different approach to designing a cloud stack that isolates application components, while supporting efficient sharing. We ask the question "if the hardware supported dynamic, low-overhead sharing of arbitrary-sized memory regions between otherwise isolated regions, how would this impact the cloud stack design?" We exploit hardware support for *memory capabilities* [23, 70], which impose flexible bounds on all memory accesses, allowing components to be isolated without page table modifications or adherence to page boundaries. This offers a new opportunity to design memory sharing primitives between isolated compartments with zero-copy semantics.

We describe **CAP-VMs (cVMs)**, a new VM-like abstraction for executing isolated components and sharing data across them. cVMs are enforced by a small TCB that uses memory capabilities to isolate and share data between compartments efficiently. Through the use of a *hybrid* capability model [66], cVMs avoid having to port application components to use

capability instructions, circumventing compatibility issues that typically plague capability architectures.

Using memory capabilities as part of a cloud stack, however, raises new challenges: the cloud stack must (i) support existing capability-unaware software without cumbersome code changes, bespoke compiler support, or manual management of capabilities across isolation boundaries; (ii) remain compatible with existing OS abstractions, e.g., POSIX interfaces, all while keeping the TCB small; and (iii) offer efficient IPC-like primitives for otherwise untrusted components to share data safely and take advantage of the potential zero-copy sharing enabled by capabilities.

To address the above challenges, cVMs make the following design contributions:

(1) Strong isolation through capabilities. Multiple cVMs share a single virtual address space safely through capabilities. Each cVM is sandboxed by a pair of *default* capabilities, which confine the accesses of all instructions inside a cVM to its own memory boundaries. To avoid having to port existing application components to a capability architecture, cVMs allow them to execute unmodified by using CHERI's *hybrid* capability architecture [66], which integrates capabilities with a conventional MMU architecture. In addition, cVMs strictly limit how CHERI capabilities can be used to avoid known capability revocation overheads: cVMs are not permitted to store or export capabilities, and the transitions of communication capabilities are controlled by a trusted component.

(2) Bespoke OS support through a library OS. cVMs are self-contained with a small TCB, reducing reliance on the external cloud stack, while providing POSIX compatibility. They include a bespoke *library OS* with POSIX interfaces for, e.g., filesystem and network operations with cryptography for transparent protection, which is protected from application code using capabilities. In the library OS, each cVM implements its own namespace for filesystem objects, virtual devices, cryptographic I/O keys etc. Only low-level resources, e.g., execution contexts for threads and I/O device operations, are shared and provided by an external host OS kernel.

(3) Efficient data sharing primitives. cVMs offer two low-level primitives to share data efficiently without exposing application code to capabilities, which are hidden behind a small, trusted *Intravisor*: (i) a *CP_File* API allows application components to share arbitrary buffers through an asynchronous read/write interface. Under the hood, the cVM implementation uses capability-aware instructions to exchange the rights to safely access each other's memory, and read/write data at byte granularity at the cost of a single memory copy (whereas traditional file-oriented IPC would require two copies); and (ii) a *CP_Call* API transfers control between cVMs, which, e.g., can be used to implement synchronization mechanisms. By combining these two primitives, higher-level APIs are possible: (iii) a *CP_Stream* API supports efficient stream-oriented data exchange between cVMs with one memory copy.

We implement cVMs on the CHERI RISC-V64 architecture, executable on FPGA hardware with CHERI support and multi-core RISC-V hardware. Our evaluation shows that cVMs provide a practical isolation abstraction with efficient data sharing: using the CP_Stream API for inter-cVM communication reduces latency for Redis by up to 54% compared to classical socket interfaces, and reduces its standard deviation by up to 2.1×. When isolating a cryptography component of a Python-based service, cVMs introduce an overhead of up to 12% compared to a monolithic baseline.

2 Hardware Isolation Support

Next we survey the design space for isolation and sharing in cloud environments in more detail (§2.1), provide background on capability support on modern hardware (§2.2), and describe our threat model (§2.3).

2.1 Isolation and sharing in the cloud

We argue that VMs and containers are two extremes of component isolation. VMs virtualize hardware interfaces such as page tables, instructions, traps, and physical device interfaces to manage both isolation and communication; containers virtualize pure software interfaces such as processes, files, and sockets for the same purposes.

Compatibility. Both VMs and containers are compatible with existing applications, which is critical for adoption in cloud environments. VMs can execute an unmodified guest OS on top of a hypervisor, making virtualization transparent to applications inside VMs. Conversely, containers execute unmodified applications on top of the same host OS kernel that manages other containerized and non-containerized applications. In both cases, OS interfaces and semantics used by the virtualized applications remain unmodified compared to a non-virtualized environment.

But the compatibility offered by these technologies lowers communication performance, which is often exacerbated as we try to achieve better isolation between components.

Isolation. Despite strict isolation between the memory of containers, there is a lack of isolation of the TCB that manages the virtualization mechanism itself. Conventional container platforms, e.g., Linux containers [2], share privileged state, as they employ namespace virtualization: the OS kernel creates separate process identifiers, devices, filesystem views etc., which offer the illusion that a process group exists in isolation. In reality, containers share kernel data structures, and privilege escalation inside one container may lead to the compromise of all containers [3,5]. In comparison, VMs are virtualized through narrower interfaces, resulting in a conceptually simpler hypervisor that is harder to compromise [15,56].

Unfortunately, stronger isolation comes at a performance price from both known hardware inefficiencies [14,41,61] as well as less flexible mechanisms for data sharing.

Sharing. Components of cloud applications typically use

networking as a means of communication. Even if multiple components are co-located on the same host, they may use a reliable network transport protocol, e.g., TCP. While this helps with scalability, it adds overhead for co-located components, making optimizations based on direct memory sharing attractive. Both VMs and containers use page-based memory isolation, which limits the performance of memory sharing: mechanisms must be aware of page boundaries to avoid leaking sensitive data, and page table modifications for on-demand sharing are known to be expensive [62].

Co-location opens up two avenues for performance improvements: (1) sharing can transparently speed up communication of co-located components [44, 47]; and (2) new communication interfaces can be tailored toward efficient sharing between components.

2.2 CHERI capability architecture

In cloud applications with many services [26], traditional network-based communication shows its performance limits between tightly-coupled components [33]. Therefore, we aim to co-locate components and design a cloud stack with efficient isolation and communication interfaces and mechanisms. This requires, however, new hardware support for isolation and sharing that is free of the "MMU tax" of page-level privileged memory protection.

Memory capabilities [18] are a protection and sharing mechanism supported by the hardware. The *CHERI* architecture [64, 70] implements capabilities as an alternative to traditional memory pointers. A capability is stored in memory or registers, and encodes an address range with permissions, e.g., referring to a read-only buffer or a callable function.

CHERI protects capabilities by enforcing three properties: (1) *provenance validity* ensures that a capability can only be "derived", i.e., constructed, from another valid capability, i.e., it is not possible to cast an arbitrary byte sequence to a capability; (2) *capability integrity* means that capabilities stored in memory cannot be modified, which CHERI achieves through transparent memory tagging [70]; and (3) *capability monotonicity* requires that, if a capability is stored in a register, its bounds and permissions can only be reduced, e.g., a read-only capability cannot be turned into a read-write one.

Building capability-based compartments. CHERI capabilities can be used to compartmentalize software components, e.g., plugins or libraries in a program, by giving each capabilities to separate memory regions. The above properties enforced by CHERI ensure that compartments can coexist in the same address space, and remain isolated as long as their initial capabilities point to disjoint data and code in memory. The application can, of course, grant each compartment extra capabilities, e.g., to allow particular cross-compartment memory accesses or function calls.

Pure- and hybrid-cap code. CHERI distinguishes between two execution modes [66]: (i) in *pure-cap* mode, all pointers must be capabilities,[1] and code must use a new set of capability-aware instructions; and (ii) in *hybrid-cap* mode, code can mix ordinary and capability-aware instructions, which allows the coexistence of capability-unaware and pure-cap code via wrapping functions. This facilitates the incremental adoption of capabilities in software.

When accessing memory, pure-cap code must use new instructions that use capability registers instead of regular registers. In addition, secure calls across capability-isolated components must use a `CInvoke` instruction, which requires a pair of capabilities: the target function address, and an arbitrary value that is meaningful to the callee function (e.g., an identifier for an object managed by the callee).

To ensure that both capabilities are used correctly by `CInvoke`, e.g., thwarting a malicious caller from passing a callee object identifier that was meant for a different callee function, the callee can "seal" pairs of capabilities together using the `CSeal` instruction. `CInvoke` only accepts correctly sealed pairs of capabilities.

Hybrid-cap code relies on two new capability registers, the *default data capability* (`ddc`) and the *program counter capability* (`pcc`), which are used implicitly by capability-unaware instructions. The OS starts all processes by setting `ddc` and `pcc` to the entire virtual address space. Capability-aware code then creates new capabilities from these registers, preserving CHERI's provenance, integrity and monotonicity properties.

Pure-cap code thus introduces compatibility challenges:

- All pointers in pure-cap code are capabilities that occupy 16 bytes instead of the ordinary 8 bytes, and must be 16-byte aligned. This decreases CPU cache effectiveness, and may require extra effort to align capability and non-capability elements in data structures.
- It is not possible to cast between addresses and various types of capability-based pointers, because CHERI distinguishes between them and imposes bounds on pointers [65]. C/C++ code that uses raw casts—a commonly found idiom in low-level system software—requires substantial modifications. For example, the strict bounds in capabilities are typically incompatible with memory allocators that place metadata before allocated data.
- While CHERI compresses capabilities, they can still result in memory bloat, because larger sizes are subject to coarser address discretization. Large allocations with capabilities may require stronger alignment and extra padding [69].
- CHERI advocates for a trusted, system-wide *garbage collector* to manage capabilities to dynamically-allocated memory [66]. It is important to ensure that allocations are not reused while valid capabilities pointing to them still exist. Since new capabilities can be derived from existing ones, and stored on the heap, stack, and in registers, all capabilities derived from an allocation must be either invalidated (i.e., revoked), or allocations cannot be reused

[1] CHERI has separate registers for regular data and capabilities.

while such capabilities are valid. A garbage collector (as opposed to expensive hardware support for capability revocation) addresses this issue, but it is a disruptive change in cloud environments, potentially leading to delays in resource reclamation and increased tail latencies.

Removing the need to use capability-aware code is important in cloud environments with limited control over tenant code. Therefore, we want to explore a design for a cloud stack that compartmentalizes application components using CHERI's hybrid-cap mode, without the disadvantages of pure capability-aware code.

2.3 Threat model

Cloud environments support multiple, isolated application components, and thus we consider attacks in which an attacker controls a malicious component that interferes with another component by probing interfaces or trying to escape its sandbox. We assume that the attacker has full control over the application components and a library OS, e.g., by exploiting vulnerabilities inside the compartment or by executing arbitrary code that includes capability-aware instructions.

Our TCB includes the underlying host OS kernel, but the entire application stack (program, libraries and library OS) is considered untrusted. We assume that the CHERI hardware implementation is correct. We do not analyse side-channel attacks against CHERI, which is an important, yet orthogonal consideration that affects both the architectural and micro-architectural levels [67].

3 cVM Design

cVMs are a new virtualisation and compartmentalization abstraction for application components. Such components can often be co-located and exchange data, and cVMs isolate them with support for low-overhead data exchange using CHERI capabilities. The design of cVMs has the following features:

Separate namespaces. Unlike containers, cVMs do not rely on a shared OS kernel for namespace isolation. They use capabilities to add a new userspace-level isolation boundary, moving OS kernel functionality from a privileged to an unprivileged layer. cVMs only use the host OS for execution contexts, synchronisation, and I/O, thus resembling VMs.

Bypassed communication. cVMs are mutually untrusted, but communication bypasses the host OS kernel for performance. They use capabilities for on-demand access to memory regions used for communication, without compromising neighbouring memory.

Low-overhead isolation. cVMs use capabilities for low-overhead isolation of both process and program modules. For example, cVMs can isolate shared libraries with minimal changes to the calling interface.

Compatibility. cVMs use CHERI's hybrid-cap mode. Capabilities are thus hidden from application code, which only needs changes to use new communication APIs.

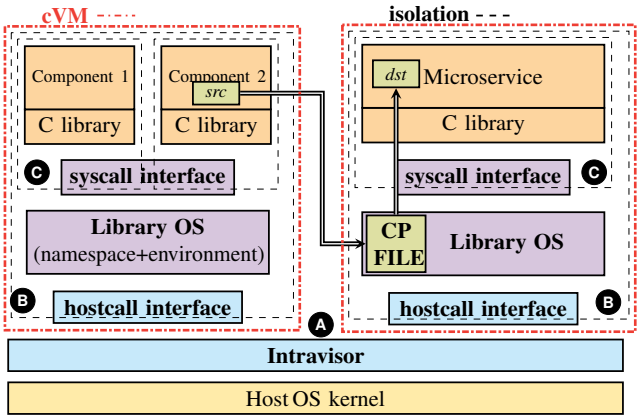


Fig. 1: cVM architecture

3.1 Architecture overview

Fig. 1 shows the architecture of cVMs. Each cVM Ⓐ is an application component, such as a process or library, and has three parts: (i) program binaries and their libraries; (ii) a standard C library; and (iii) a library OS.

cVMs add two new isolation boundaries, enforced through capabilities. The *Intravisor boundary* Ⓑ separates the *Intravisor* from all cVMs, and cVMs from each other. The Intravisor is responsible for the lifecycle and isolation of cVMs, allows safe communication between them, and provides other primitives that cannot be implemented inside the unprivileged library OS (e.g., storage and networking I/O, time, threading and synchronisation). It has access to the memory of all cVMs, but not the other way around.

The *Program boundary* Ⓒ separates programs from the library OS that provides them the namespace for all OS primitives. A single library OS instance can thus host multiple, mutually-isolated programs with their own code and data (left-most cVM in Fig. 1).

These isolation boundaries are enforced by CHERI capabilities; compartmentalized content cannot access memory beyond its boundary, except through the controlled interfaces described next. Finally, there is a classical separation from the host OS, using CPU rings and MMU-based isolation.

3.2 Isolation boundaries

We now describe how cVM are isolated in more detail (see Fig. 2). Each program compartment contains the code and data of its binary, its dependencies (shared libraries), and the standard C library; the cVM also contains the library OS, which provides the OS functionality.

Isolation boundaries are enforced by giving each its own default CHERI capabilities using the `pcc` and `dcc` registers (see §2.2) with non-overlapping address ranges; compartmentalized code thus cannot load, store or jump into memory outside that granted by the capabilities that it holds. To allow ❶ program → libOS and ❷ libOS → Intravisor calls, cVMs use extra capabilities that grant controlled access to functions outside the respective compartment.

Tab. 1: cVM API

Type	API function	Description
Creation	`cp_cvm_make(cp_config_t *cfg, char *libos, char *disk.img, int argc, char *argv[])`	Create new cVM
CP_File	`cp_file_make(char *key, size_t key_size, void *addr, size_t size)`	Make CP_File for buffer `addr` & publish with `key`
	`cp_file_destroy(int file)`	Destroy CP_File
	`cp_file_get(char *key, size_t key_size)`	Get CP_File with `key` from another cVM
	`cp_file_read,cp_file_write(int file, char *key, size_t key_size)`	Read/write data via CP_File `file`
	`cp_file_wait,cp_file_notify(int file)`	Wait/notify signal via CP_File `file`
CP_Call	`cp_call_make(char *key, size_t key_size, void *func)`	Make CP_Call for `func` & publish with `key`
	`cp_call_destroy(int call)`	Destroy previously created CP_Call
	`cp_call_get(char *key, size_t key_size)`	Get CP_Call with `key` from another cVM
	`cp_call(int call, bool async, void *arg, size_t size)`	Call CP_File `call` with arguments
CP_Stream	`cp_stream_make(char *key, size_t key_size)`	Make CP_Stream & publish with given `key`
	`cp_stream_destroy(int stream)`	Destroy CP_Stream
	`cp_stream_get(char *key, size_t key_size)`	Get CP_Stream with `key` from another cVM
	`cp_stream_send(int stream, void *buf, size_t size)`	Send buffer through CP_Stream
	`cp_stream_recv(int stream, long id, void *buf, size_t size)`	Post buffer to receive through CP_Stream.
	`cp_stream_poll(int stream, long *id, size_t nid, int timeout)`	Poll for data on receive buffers of CP_Stream

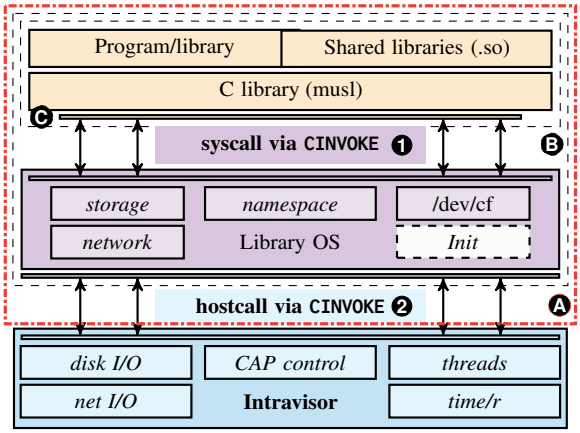


Fig. 2: Anatomy of a cVM

cVMs need to implement the equivalent of user/kernel separation using CHERI capabilities in userspace. When loading a program, a set of capabilities is therefore given to the syscall handler functions of the library OS. The standard C library uses these capabilities to invoke system calls on the library OS through the `CInvoke` instruction, while the rest of the application remains capability-unaware. The library OS has full access to the programs that it manages.

cVMs also need to implement the equivalent of guest/host (or VM/hypervisor) separation using CHERI capabilities in userspace. When creating a cVM, the Intravisor installs capabilities to its own host system call handlers on the new library OS instance; in turn, the library OS uses `CInvoke` to invoke Intravisor operations.

3.3 Creation and communication API

cVMs combine compatibility and flexibility when isolating cloud services. They support the execution of complete application components using a process isolation abstraction, but also that of individual library components.

Tab. 1 shows the cVM API. New cVMs are created by `cp_cvm_make()`; similar to `fork()`/`exec()`, it accepts a disk image file, a program binary to load into the cVM, and a function in that binary to launch. If a cVM isolates a standalone library, `cp_call()` invokes functions in the library.

cVMs use CHERI capabilities for efficient inter-cVM communication. The Intravisor exchanges an initial set of capabilities between cVMs to allow communication.

CP_File. This primitive introduces a file-like API to access memory from another cVM at arbitrary granularity; the use of capabilities in CP_File permits bypassed access to memory without repeated mediation by the Intravisor.

A *donor* cVM registers a memory region with the Intravisor to share with other cVMs via `cp_file_make()`; a *recipient* cVM calls `cp_file_get()` with the same key to obtain access. The cVMs then access data in the memory region via `cp_file_read/write()`. Internally, the library OS uses capability-aware code to copy data directly between the cVMs (using `capcpy`; see §4).

To support asynchronous data transfers, `cp_file_wait()` and `cp_file_notify()` allow callers to wait for and notify events on a CP_File, respectively. Finally, the donor cVM calls `cp_file_destroy()` to destroy it, revoking all access.

CP_Call. This primitive invokes functions outside the calling cVM, e.g., a callback function in the library OS, or a function in a shared library. cVMs manage CP_Calls as follows: `cp_call_make()` registers a function in the donor that recipients can look up using `cp_call_get()` and then call with `cp_call()`. The call is received by the Intravisor, which creates a new thread in the donor's cVM, sets it to execution to the target function with given arguments and, optionally, waits for its completion, based on the `async` argument.

CP_Stream. By composing the CP_Files and CP_Calls APIs, it is possible to construct more complex communication mechanisms. For example, we have built a stream-oriented API for inter-cVM communication in which the sender does not need to know where data is copied.

A recipient cVM calls `cp_stream_recv()` to register buffers for incoming messages (internally, a list of CP_Files); a sender cVM calls `cp_stream_send()` to copy data into any of the buffers available in the recipient. The recipient is then informed of data transfers when calling `cp_stream_poll()`.

3.4 Capability management

The use of CHERI capabilities introduces two problems that cVMs must avoid: avoiding the need for application code to become capability-aware and performance problems when revoking capabilities.

As explained in §2.2, making an application fully capability-aware requires code changes. The design of cVMs avoids this by limiting the use of capability-aware code to a small portion of the standard C library, the library OS and the Intravisor, which explicitly handle the CP_Files and CP_Calls abstractions through syscall trampolines.

In the cVM design, we want to avoid centralized trusted mechanisms for capability revocation (see §2.2), as this goes against our goal of minimizing overheads and TCB size. Therefore, only the Intravisor is permitted to store CHERI capabilities in memory: all capabilities that are passed by the Intravisor to cVMs have the `CAP_STORE` permission withheld. Instead of having to perform expensive garbage collection, revocation can now be done by clearing a small number of capability registers. This can be done efficiently when programs call the cVM API to avoid interrupting execution.

4 Implementation

Next, we report implementation details of cVMs on the CHERI RISC-V64 platform. Our implementation consists of 5,200 lines of C code and 100 lines of assembly for the Intravisor, and 1,800 lines of C code and 200 lines of assembly for the Init service, the Hostcall interface and CAP Devices. It uses the Linux Kernel Library (LKL) v4.17.0 [36] as the library OS and the musl standard C library v1.2.1 [42]. As the host OS kernel, we use CheriBSD [25].

4.1 cVM lifecycle

Initialisation. The boot process of a cVM is trigged by the Intravisor. It receives a deployment configuration for the cVM, which includes the heap size, the disk image location, the permitted interfaces, etc. It also defines the version and location of an Init service (see below) and the library OS binaries. The Intravisor first allocates memory for the cVM binary, stack and heap. It also allocates memory for the thread stack pool. Our implementation of cVMs cannot change the size of heap and stack at runtime, but this is a minor limitation given the size is in terms of virtual memory, and is only committed to physical memory on demand. Just as cloud providers prefer re-instantiating VMs over the use of memory ballooning, we expect large resource size changes to re-instantiate cVMs.

All threads must be created inside a compartment's memory, thus the Intravisor pre-allocates memory for future thread stacks. After that, the Intravisor deploys the image of the Init service into the cVM and spawns the initial thread in the context of the cVM. This thread prepares the hostcall callback tables, and enters the cVM via the `CInvoke`-based interface created by the Intravisor.

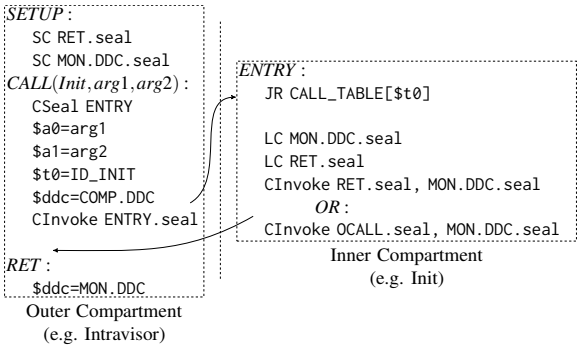


Fig. 3: ICALL and OCALL implementation

The Init service (see Fig. 2) is responsible for initializing all components at deployment, and creates the communication interface between the library OS and the host system. It is part of the library OS isolation layer, which means that it can access the memory of the application component. It initialises the library OS, builds the syscall interface for the program (or library), deploys its binary and calls the entry function (e.g., `c_start()`). For an executable binary, it launches the program; for a library, the entry function initializes a CP_Stream and registers the public library functions with the Intravisor.

Execution. cVMs use the Linux kernel library (LKL) [36] as a library OS that provides a Linux-compatible environment. LKL processes system calls and requests the host OS kernel to perform actions as needed.

LKL's storage and networking backends implement lean interfaces for hardware I/O devices: disk I/O has three hostcalls (`disk_read/write()`, `disk_getsize()`); networking uses only `net_read/write()`. The `disk_read/write` functions are applied to a file descriptor of the disk image; the network functions are invoked on a TAP device. The remaining functions in the hostcall interface are straightforward: they offer support for time and timer functions, debug output, threading and locking, and management of CAP Devices (see §4.3).

Threading. For simplicity, cVMs use a 1-to-1 threading model. When a cVM creates a thread, the `pthread` library requests an execution context from LKL, which in turn, requests a new thread from the host OS kernel. This requires the integration of the `pthread` implementations inside the cVM and the host—both must maintain their own thread-local stores, pointers to `thread_structs`, etc.

When LKL requests a thread, it prepares a structure with an address of the entrance function, and a pointer to the arguments. This is passed to the host OS kernel, and the Intravisor creates a new thread with the provided arguments: it allocates a stack for the thread from the thread stack pool, pre-allocated at boot. After that, the new thread is ready to enter the cVM using `CInvoke` and capabilities are created by the hostcall interface. Prior to entering, the Intravisor switches the thread pointer `tp` register. Inside a cVM, threads have LKL TP values; when processing hostcalls, they have host ones.

4.2 Calls between nested compartments

cVMs use the `CInvoke` instruction to call functions between isolation layers, both (i) from an outer to an inner layer (ICALL), e.g., when the Intravisor invokes Init; and (ii) from an inner to an outer layer (OCALL), e.g., when performing a syscall or hostcall.

`CInvoke` takes two *sealed* capabilities (see §2.2) as arguments: (i) one with a new Program Counter Capability (`pcc`) value and another that points to a memory region that becomes accessible after the instruction execution. The `pcc` is replaced by the first unsealed capability; the second capability moves to the `ct6` (`C31`) register in the unsealed form.

Next, we explain how `CInvoke` is used to implement both ICALLs and OCALLs:

ICALLs. Fig. 3 shows the switching mechanism for ICALLs. In this example, the Intravisor in the outer layer calls Init in the inner layer. To make the call, the caller prepares the first capability that points to the entry point inside the compartment. This capability, together with the corresponding data capability, defines the default capabilities of the inner compartment. Inside the compartment, these capabilities, `COMP.DDC` and `ENTRY.PCC` become `ddc` and `pcc`, respectively. While the `ENTRY.PCC` capability can be passed as the first argument of `CInvoke`, `COMP.DCC` must be loaded by the caller prior to switching (see Fig. 3).

To return from the compartment or grant permission to invoke functions in the outer layer from the inner layer, further capabilities are needed: these are stored in memory by the Intravisor before `CInvoke` is called, in a structure that we call the *Affix*. They include a sealed `ddc` of the outer layer (`MON.DDC.sealed`). Without this capability, the Intravisor could not change `ddc` from the inner to the outer layer on return in order to access the Intravisor's data. This capability can only be fetched from the inner layer—the accessible memory is restricted by the `ddc` of the inner layer.

The Affix also includes `RET.sealed` and `OCALL.sealed`, which are two sealed `pcc` capabilities to entry functions in the outer layer. The former is used to return from the compartment; the latter points to an entry function, which is used when the inner layer calls a function of the outer layer (e.g., `print()`) and returns to continue execution inside the compartment. This is used for the syscall and hostcall interfaces. Capabilities in the Affix are created by the Intravisor and stored on the stack and inside per-compartment private stores.

OCALLs share many similarities with ICALLS. The caller prepares a sealed capability of the return address. After the end of a function, the callee uses `CInvoke` and the execution of the caller continues from the desired address. Together with `CInvoke`, the callee passes the sealed capability `MON.DDC.seal`, which was passed originally inside the Affix. It is put into `ddc` after the function returns.

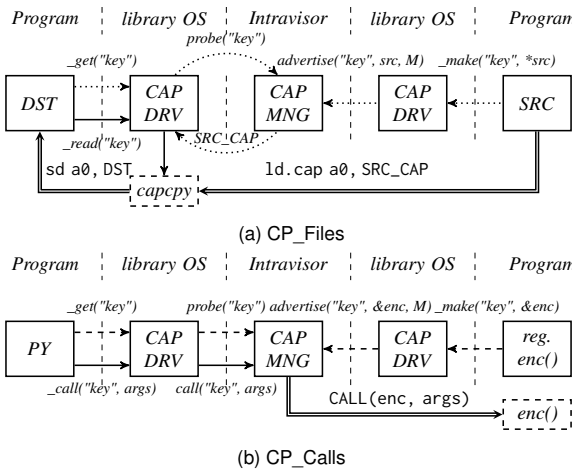


Fig. 4: Implementation of communication mechanisms

4.3 Communication mechanisms

The data sharing API between cVMs from §3.3 is also based on capabilities. Data referenced by capabilities, however, can only be manipulated by capability-aware instructions, which do not exist in native code. To resolve this issue, we *mediate* the interaction between hybrid-cap code and capabilities using virtual devices called *CAP Devices*.

The CP_Files, CP_Calls, and CP_Streams primitives are implemented using character devices, which are created by the library OS and Intravisor. A program can read/write from/to these devices, and the corresponding operations are performed by capability-aware code inside drivers.

This design has two advantages: (i) despite its one memory copy, it is faster than traditional communication interfaces (see §6.5); and (ii) it supports a simple mechanism to revoke capabilities. A remote cVM can inform the Intravisor of the revocation, which then requests the library OS to destroy the corresponding CAP Device. To revoke capabilities in pure-cap code, a Intravisor would have to stop the cVM execution and destroy capabilities manually.

CP_Files support regular POSIX file operations. In contrast to ordinary files, the content of CP_Files is not cached by the page cache, and read/write operations can be unaligned.

Fig. 4a shows the implementation. A donor cVM advertises one or more memory regions defined by *keys*, and a recipient cVM probes the Intravisor for a given key. The Intravisor verifies the access control list and builds a CAP Device for the target CP_File (e.g., `/dev/cf0`). For the donor cVM to revoke access, it uses its own CAP Device to request revocation, and the Intravisor, together with the library OS, destroy the CP_Files (`cf0`) driver along with its capabilities.

When the recipient cVM issues a `cp_file_read()` call, the driver uses *capcpy* to copy data. For `cp_file_read()`, it uses `ld.cap` to read data from a remote cVM and store it via `sd`; a `cp_file_write()` does the reverse.

CP_Calls. To expose a function, a cVM creates an ICALL entry and registers it with the Intravisor (see Fig. 4b). The In-

travisor maintains a table of exported functions for each cVM, called cVM-RPCs. It consists of access control records with capabilities, name identifiers and permissions. Application components interact with the cVM-RPCs via CAP Devices, a management interface (`/dev/cf`), and the Intravisor.

Any function can be invoked by CP_Calls including ones inside the library OS. This enables the use of CP_Calls as a notification mechanism between CP_Files. The donor blocks execution until the recipient cVM reads data. It makes the `wait()` call with the driver, the driver puts the execution thread in the work queue and waits for the signal. Prior to blocking, it registers a wake-up CP_Call with the Intravisor. The recipient cVM, in turn, finishes its operations with the CP_Files, and notifies the donor via this CP_Call.

These basic operations can be composed to create higher-level protocols, and a single CAP Device can handle multiple memory regions. For example, for Redis (see §6.3), we use a series of read/write operations with a single notification as well as batched reads with different capabilities.

CP_Streams. In contrast to CP_Files, when sending data, the destination for CP_Streams is unknown, and `cp_stream_send()` only knows the source. Therefore, one side of the communication pre-registers one or more destination buffers via `cp_stream_recv()`, and uses `cp_stream_poll()` to block. The remote side uses CP_Call to enter the remote compartment, atomically fetches one destination buffer from a pre-registered queue of buffers, and copies into this buffer data via `capcpy`. It then wakes up the poll queue and returns.

Hostcall Interface. The Intravisor does not impose restrictions on the number of calls in the hostcall interface. For the LKL library OS, the Intravisor provides 24 hostcalls for minimal operation. In addition, 2 hostcalls are necessary for disk I/O, 3 for network I/O, and 10 for the capability-based communication primitives.

4.4 Capability revocation

Data transfers (`capcpy`) are performed by the drivers of CAP Devices without direct involvement of the Intravisor, which enhances performance and reduces the TCB. This, however, means that the driver must have access to the capabilities provided by the donor. We do not consider the driver trusted, thus it may be compromised by an adversary who obtains access to capabilities and memory outside the cVM after the end of a communication session. To mitigate against this threat, cVMs support a revocation mechanism. It guarantees that, once the donor cVM revokes capabilities, they are destroyed, and a recipient cVM cannot use them.

First, cVMs or communication capabilities are not created with the `PERMIT_STORE_CAP` permission. Code inside a cVM thus cannot store capabilities to memory: it can load them, modify, create new capabilities, but it fails on `ST`. The communication capabilities are stored once by the Intravisor, when the communication is established, and destroyed at the end. Second, the revoked capabilities in the CPU context are destroyed after a context switch by the host OS kernel.

5 Security Analysis

According to our threat model from §2.3, an attacker can gain control over a cVM. However, we guarantee that they cannot escape the compartment or access memory beyond its boundary due to the CHERI architectural properties (see §2.2): the ddc and pcc capabilities always apply, are non-extensible, and are controlled by the Intravisor.

Hybrid-cap code may be vulnerable to attacks that attempt to break execution flow. An adversary may inject capability-aware instructions (e.g., `CLD`/`CSD`, `CInvoke`) to access data and code outside of the compartment. To do this, the adversary requires capabilities, which they cannot construct from the available data inside a cVM.

To escape a compartment, an adversary must obtain appropriate capabilities. Each cVM, however, only maintains a few capabilities: a compartment (i) receives three sealed capabilities via Affixes, which can be inspected by an adversary but not unsealed to create new capabilities; and (ii) may receive capabilities used by CP_Files and CP_Streams. These capabilities can be exploited by an adversary after gaining full control over the library OS. Since these are data capabilities, they cannot be used to create code capabilities, which are needed to escape the compartment. The adversary also cannot store these capabilities due to their permissions. Finally, they also cannot be exported outside of the compartment via the hostcall interface, because the interface does not handle capabilities and instead corrupts them.

Hybrid-cap code may contain security flaws, but an adversary cannot escape confinement, unless a flaw in the outer level provides them with unsealed capabilities. In our design, this is unlikely due to the Intravisor's small TCB. The adversary cannot export or import capabilities via the hostcall interface or use them beyond a communication session. Vulnerable hybrid-cap code cannot abuse host system calls, escalate privileges or attack other cVMs, because the host OS kernel ignores all direct system calls from cVMs.

cVMs are intra-process compartments that share micro-architectural state and rely on the correctness of the CHERI architecture, which does not have special mechanisms to prevent side-channel attacks. Nonetheless, there are plans for CHERI to include explicit compartment identifiers (CIDs) in a future version of the architecture [67]. This will ensure that sensitive micro-architectural state is appropriately tagged by each cVM, similar to tagged TLB entries. This can be used to prevent attacks, such as training the branch predictor by one cVM to direct speculative execution in another cVM.

6 Evaluation

We now explore the performance of cVMs and the proposed communication interfaces. We begin with an overview of our evaluation platforms and workloads (§6.1). We then compare

the performance of applications deployed with cVMs and Docker containers (§6.2). In §6.3, we validate the efficiency of inter-cVM communication mechanisms; in §6.4, we explore the use of cVMs for component compartmentalisation; and in §6.5, we compare inter-cVM communication mechanisms with existing OS mechanisms. Finally, §6.6 explores the deployment performance of cVMs and Docker containers.

6.1 Experimental environment

The CHERI architecture is under active development and, while ARM's Morello board with CHERI support has been announced [9], it is unavailable at the time of writing. Therefore, we use two **evaluation platforms**: (1) a single-core FPGA-based CHERI implementation [21]; and (2) a multi-core SiFive RISC-V implementation without CHERI support.

FPGA CHERI. We synthesize an FPGA image from DARPA's CHERI FETT program [22] (`agfi-026d853003d6c433a`), that ships with a single-core RISC-V64 CHERI system based on the FLUTE core (5-stage, in-order pipeline, running at 100 MHz) [49], and execute it on AWS F1 [8]. We use CheriBSD as the host OS kernel, compiled as a hybrid-cap system with LLVM v11.0.0 and cheribuild [16].

The FPGA implementation enables a quantitative evaluation of cVMs, but has limitations: (i) it has a single-core CPU with low clock frequency; (ii) its peripheral devices, in particular storage devices, are emulated by the host; and (iii) DRAM latency is disproportionately low compared to the CPU clock speed. As a consequence, we cannot realistically execute typical cloud workloads that are memory- and I/O-bound and use multiple CPU cores. We also cannot eliminate system noise by pinning tasks to separate cores.

SiFive RISC-V. To avoid the abovementioned limitations, we also evaluate cVMs on a HiFive Unmatched RISC-V board [30], which has 4 RISC-V64 (dual-issue, in-order) CPU cores running at 1.2 GHz. The CPU does not have CHERI support, and we instead replace all CHERI instructions with their native RISC-V versions. Our applications execute on Ubuntu v20.04 with Linux v5.11.0 and the RISC-V Docker port [48] with Alpine containers [7]. Our IPC micro-benchmarks execute on FreeBSD 14, as the FPGA version uses CheriBSD, and we run them on both platforms.

This approach allows us to execute realistic cloud applications. We run CHERI-equivalent code and data paths while remaining compatible with existing RISC-V platforms (e.g., by replacing capability loads/stores with ordinary `ld`/`st` instructions, `CInvoke` with `jr`, etc.). Note that security is therefore not enforced.

Application workloads. We explore cVMs using several cloud applications and micro-benchmarks to evaluate their performance and isolation requirements:

NGINX/Redis (§6.2). This is a two-tier microservice deployment that evaluates the YCSB benchmark [72] using the NGINX [43] web server and the Redis [46] key/value store. NGINX acts as an API gateway and translates REST requests into Redis queries. When co-located, these services have a substantial amount of communication between them. We demonstrate that the cVMs interfaces, CP_Files and CP_Streams, significantly reduce overhead, using the SiFive platform to compare cVMs against a deployment using Docker containers [40].

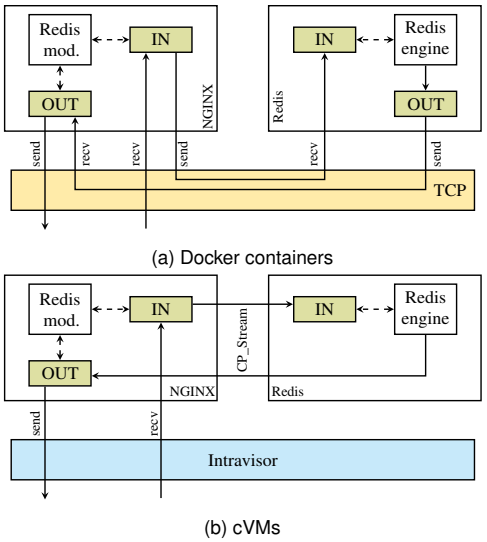


Fig. 5: Control/data flow in multi-tier deployment (NGINX/Redis)

Redis (§6.3). We execute a single-core Redis instance [46] and measure the latency of fixed-size GET and SET operations, comparing sockets and the equivalent cVM interface with CP_Streams. This experiment validates our previous results by also comparing the FPGA and SiFive environments.

Python/Library (§6.4). We measure the cost of using cVMs to isolate the components of a simple cryptographic application in Python, by deploying the Python runtime [58] and the PyCrypto cryptographic library [1] in mutually isolated cVMs that use the CP_Call and CP_File interfaces to communicate. This experiment runs on the FPGA environment.

6.2 Multi-tier deployment with NGINX/Redis

First, we compare the benefits of using cVMs when co-locating communicating components, compared to a traditional deployment with Docker containers [40].

The computational limitations of our FPGA and SiFive platforms make it unfeasible to execute a complete microservice benchmark suite such as DeathStarBench [26]. Instead, we deploy a representative YCSB benchmark [72] (workloadb; 1 KB records; read/update ratio of 95%/5%) on the SiFive platform with two-tiers: the NGNIX web server [43] acts as an API gateway that redirects incoming HTTP requests to the Redis key/value store [46], which acts as a cache for frequently used data. We use *wrk2* [6] to generate NGINX requests over a 1 GbE network, measuring the latency of different configurations (10 connections; 4 I/O threads).

The application components benefit from co-location due to the frequent interaction between the (NGINX) API gateway

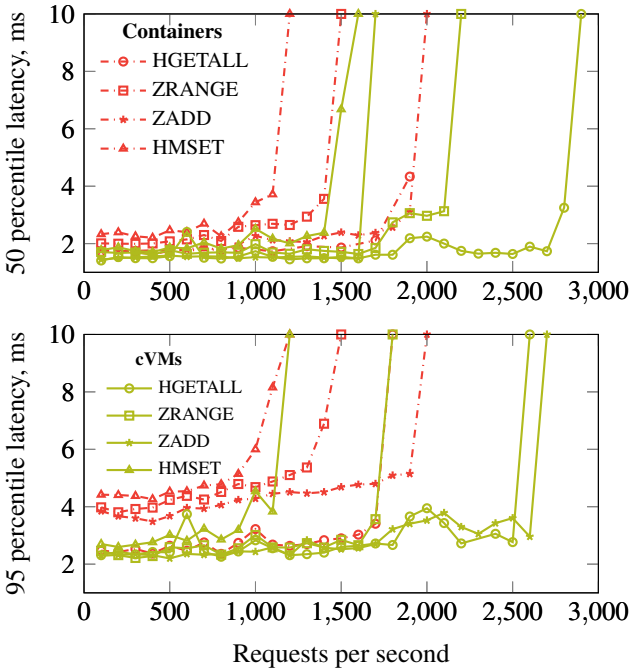


Fig. 6: Multi-tier deployment performance (NGINX/Redis)

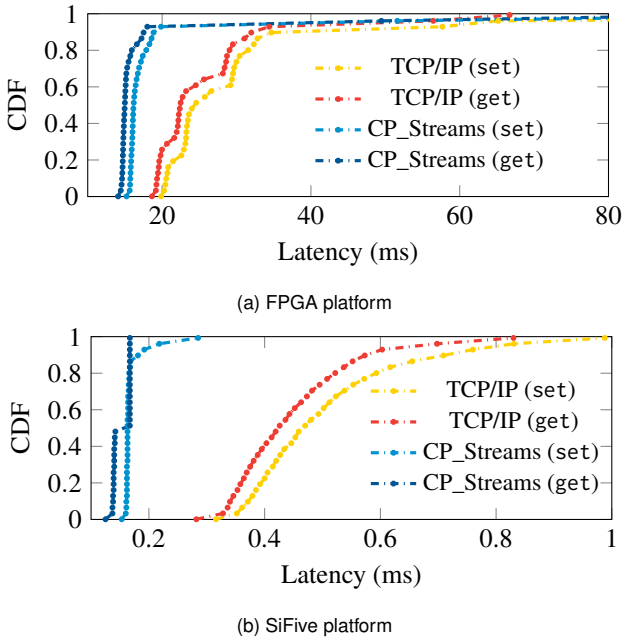


Fig. 7: Latency CDF for Redis (platform validation)

and its (Redis) cache. Fig. 5 compares the Docker and cVM deployments. Docker incurs multiple data copies between the components and the TCP/IP network stacks. As Fig. 5a shows, Redis copies values into a send buffer that is passed to the TCP/IP stack, which NGINX copies into an output buffer that is, in turn, passed to the client's network stack (for a total of 4 copies, including the kernel's TCP/IP stack).

In contrast, cVMs reduce the number of copies. Fig. 5b shows that the CP_Stream primitive requires only 2 copies: Redis values are always copied directly into NGINX's output buffer. To support this optimization, NGINX and Redis must replace their use of sockets with CP_Streams. NGINX registers the output buffer with a CP_Stream, and the CP_Stream write in Redis uses capabilities to copy data directly into the output buffer, which NGINX can then send to the client.

Fig. 6 shows the median and 95th percentile latencies for the 4 YCSB queries under various throughput regimes, comparing the baseline Docker deployment with cVMs. We can see that cVMs are more efficient: they have lower latencies in all cases (20–40% for median latency), and substantially higher throughput, with send latencies below 5 ms (33–50% for median latency).

Conclusion. In a typical deployment with multiple application components, cVMs can achieve isolation while lowering latencies and increasing throughput compared to containers. This performance gain is due to a reduced number of memory copies (via CP_Stream), using fast calls to the capability-hiding TCB in cVMs (via CP_Call within CP_Streams). Furthermore, cVMs come with a smaller TCB compared to containers. We also expect cVMs to outperform VMs because of VMs' higher overheads caused by memory virtualization (especially for memory-bound applications) and communication mechanisms (e.g., extra data copies by the guest OS and/or hypervisor, or cross-VM copies via PCIe with directly assigned devices).

6.3 Platform validation with Redis

We now validate our results by comparing the FPGA and SiFive platforms. We use Redis with a single connection that measures the latency of 1000 GET or SET operations with fixed-size keys (1 byte) and values (100 bytes). We use a simple client application that is co-located with the Redis instance. The baseline system uses separate processes and TCP/IP sockets; we use separate cVMs for each application and CP_Stream for communication (similarly to §6.2).

Fig. 7 shows the latency distribution of the GET and SET requests for all configurations. The results indeed validate our observations from the multi-tier YCSB benchmark in §6.2. cVMs exhibit lower latencies with less deviation on both platforms, compared to a native system with TCP/IP sockets: 90% of cVM requests take 14–19 ms; the baseline takes 19–35 ms on the FPGA platform. The SiFive platform supports the same conclusions, albeit with different absolute numbers. This is because the FPGA device runs at a lower clock frequency, and two processes must be co-scheduled on the same core (with both the baseline and cVMs).

Conclusion. The CP_Stream primitive in cVMs shows better performance on both the FPGA and SiFive platforms, achieving lower communication latencies across the whole throughput spectrum. We thus conclude that our end-to-end evaluation in §6.2 is representative of how cVMs would perform on a real-world CHERI-enabled CPU. In §6.5, we re-validate this by comparing cVMs against IPC primitives on all platforms.

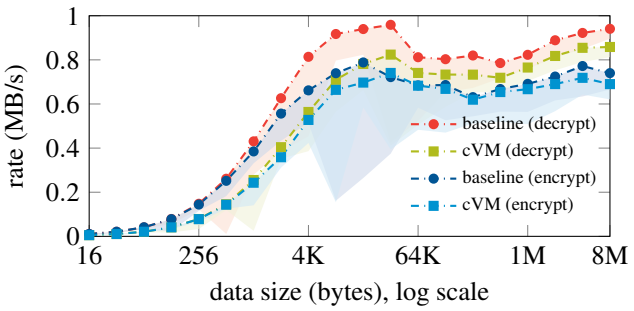


Fig. 8: cVMs with Python (AES cryptographic performance)

6.4 Process compartmentalization with Python library

Next, we explore the overhead of compartmentalizing a shared library with cryptographic operations in Python. In this case, we harden the security of a cloud application by mutually isolating the Python runtime and a native cryptographic module, PyCryptodome [1]. By using separate cVMs, we can safeguard the application against malicious interference by package managers [59], or protect the library against unauthorized access to its cryptographic keys [4].

Python creates CP_Files for the input/output buffers that it passes to the PyCryptodome library, and it uses CP_Call to transfer control to the library, using the CP_Files as arguments. (The original version instead passes raw buffer pointers.) PyCryptodome then uses these CP_Files to read its input and encrypt/decrypt it into the output buffer(using AES-128. Finally, it uses CP_Call to return execution to Python.

Fig. 8 shows the average throughput for encryption/decryption with different buffer sizes for cVMs, using the FPGA platform, and the baseline (non-isolated) system. Note that the low absolute numbers and variance (shown as shaded areas) are due to the platform limitations (single core), described in §6.1. The results in §6.3, however, show the same trend on a platform without these limitations.

We observe that cVMs have a negligible performance impact. Throughput grows until its peak with 32 KB buffers, where the encryption/decryption rates of cVMs are only 7% and 12% lower than the baseline, respectively. This amounts to 0.79 MB/s and 0.96 MB/s for the baseline, and 0.74 MB/s and 0.85 MB/s for cVMs, respectively. As expected, these overheads become even smaller as the buffer sizes grow.

Our experiment shows that CP_Call and calls into the Intravisor are reasonably efficient. For reference, the mean execution time for the AES cryptographic code with a 16 byte buffer is comparable to the time for a C binding invocation in Python. At such sizes, CP_Call invocations account for half of the overhead, which is at 97% and 101% for encryption and decryption, respectively, only slightly above a C binding invocation. The overhead reduces to 7% with larger buffers.

Conclusion. cVMs is effective at hardening applications by isolating some of their components, such as shared libraries. The required changes are minimal and do not change the semantics of the application interfaces, because the CP_File and

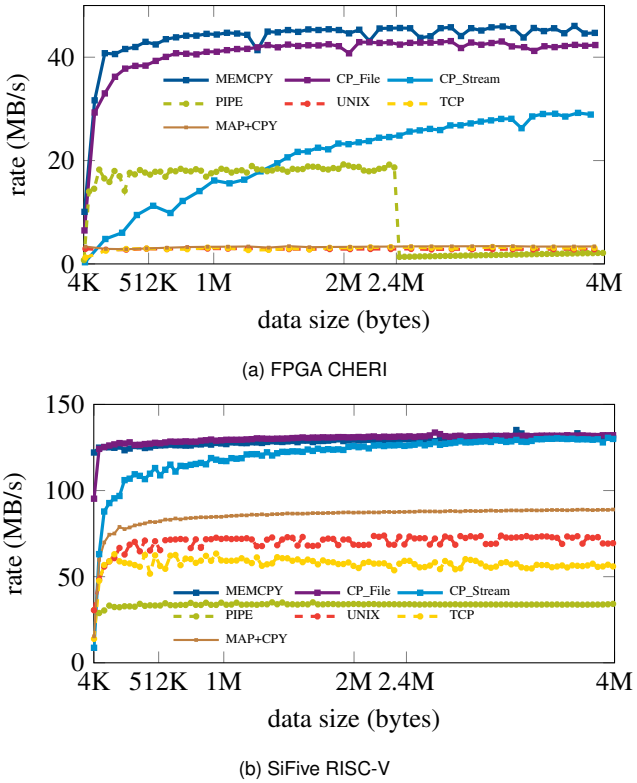


Fig. 9: Comparison of communication mechanisms

CP_Call primitives follow well-understood memory copy and function call semantics. Note that CP_Streams are constructed on top of these. The cost of this extra isolation is small, even for small buffers, and it becomes negligible as the amount of work performed between cVMs-enabled operations increases.

6.5 Inter-cVM communication

We compare cVMs to other IPC primitives in a baseline system, and re-validate our performance results across our two platforms (FPGA and SiFive). The baseline system uses two threads in a single process instead of cVMs; otherwise the FPGA implementation shows low TLB performance. We measure the performance of CP_Files and CP_Streams, pipes (`PIPE`), unix sockets (`UNIX`), TCP/IP sockets (`TCP`) and a combination of `mmap+memcpy+munmap` (`MAP+CPY`). For comparison, we also consider a raw local `memcpy` (`MEMCPY`; 4 instructions; aligned data; double-word load/store operations) as an upper performance bound. We do not evaluate CP_Calls due to the lack of an equivalent operation in the baseline kernel.

Fig. 9 shows the results under different buffer sizes on both the FPGA and SiFive platforms. First, the peak performance of `MEMCPY` on the FPGA platform is limited and fluctuates due to the TLB size and simple indexing function of its Flute CPU—these issues carry onto the other primitives, too.

The overhead of CP_Files is 6% compared to `MEMCPY` on the FPGA platform and negligible for SiFive; it significantly outperforms all baseline IPC mechanisms. This is because we do a simple cross-cVM `memcpy` using CHERI's `ld.cap` and

`cincoffsetimm` instructions to perform the memory access and to increment the capability offset, respectively. The results also show that domain transitions via `CInvoke` are efficient, as every CP_File operation requires one capability call and its return (user→library OS, and back).

All baseline IPC primitives have 2× overhead or more, because they perform more data copies than `MEMCPY` and CP_Files, closely following ideal performance. Interestingly, CP_Streams have worse performance on the FPGA platform, despite the lower number of copies, whereas they show performance close to CP_File on the SiFive platform. This is because CP_Streams offer an asynchronous communication primitive in which two concurrent processes time-share a single CPU core on the FPGA platform when using the cVM API. For the same reason, all IPC primitives have lower relative performance on the FPGA platform compared to SiFive.

UNIX sockets are the closest to CP_Streams, because both are bi-directional, support more than two parties, and have sequenced packet modes. They exhibit only 10% and 54% of the performance of CP_Streams for 4 MB buffers on the FPGA and SiFive platforms, respectively. Here, the impact of MMU manipulation can be seen: the combination of memory copies and remapping reaches 3.4 MB/s and 89 MB/s on the FPGA and SiFive platforms, respectively. This mechanism lacks a notification primitive, and, compared to CP_Files, it is 15× and 1.5× slower on each platform, respectively.

Conclusion. For a multi-core CPU architecture with CHERI, we would expect the results to be close to those of the SiFive platform, with a minor performance decrease, similar to the difference between `memcpy` and CP_Files in Fig. 9a. This potential performance degradation is significantly smaller than the measured improvements: they range between 2× for the multi-core SiFive platform against the best baseline primitive, and 2× to an order of magnitude for the single-core FPGA platform, depending on the mechanism and buffer size.

6.6 Deployment time

We compare the deployment time of cVMs with that of Docker containers. We create a Docker image with a simple "hello world" program and measure the time to execute it using a cVM and a container. For the cVM, we use a debug-free binary with the LKL library OS and the musl standard C library (≈30 MB in size) and a 10 MB application disk image. We measure two intervals, averaged over 5 runs: from the start until the output of the program, and until its termination.

On average, the Docker container requires 1.9 s to produce the output, and 2.8 s until container termination. The times for the cVM deployment are comparable, which demonstrates their low overhead: 1.7 s and 2.6 s, respectively.

7 Related Work

Intra-process compartments. Various projects apply intra-process isolation or introduce isolation primitives. CubicleOS [52] isolates components of a user-level library OS using Intel MPK; unlike cVMs, it cannot readily and efficiently support legacy POSIX calls. Shreds [20], Janus [28], Erim [60], Hodor [29], and Donkey [53] use page tag-based isolation (ARM Domains, Intel MPK, or a custom RISC-V implementation) to implement protection domains and communication. In cases in which tags can be manipulated directly by user code, e.g., using MPK's `wrpkru` instruction, the system requires a trusted toolchain or program verifier, unlike cVMs. Page tags also limit the number of compartments and communication buffers, as well as their granularity, which is not a problem for cVMs with capabilities.

NaCl [73] and WASM [27] face similar problems, as they require obsolete Intel segmentation and/or proof-carrying code that must be verified by a toolchain or loader. ConfLLVM [12] also uses MPK to isolate code inside a process, but only supports two domains with asymmetric data exchange: trusted code can only interact with untrusted code. cVMs do not limit the number of protection domains, and inter-cVM communication is symmetric.

LwCs [37] are an OS abstraction for intra-process protection, but they have page granularity, and switching domains comes at the cost of switching page tables. XFI [24] provides fine-grained memory protection and control flow integrity by extending software-based fault isolation (SFI), but SFI incurs runtime overheads and is error-prone due to its complexity.

Compartmentalisation frameworks. cVMs allow the deployment of isolated shared libraries. Prior work proposes frameworks for such compartmentalization: Wedge [11] identifies code parts that can be isolated; PrivTrans [13] is a source-code partitioning tool that separates trusted and untrusted components; Glamdring [35] does the same for trusted execution. These approaches are orthogonal to cVMs, and they could be used to generate application components.

Trusted execution. Intel SGX [31, 38, 39] provides *enclaves* as an intra-process isolation primitive. Enclaves are part of processes and cannot be accessed by privileged software or other enclaves. Frameworks, such as Graphene-SGX [19], SGX-LKL [45], Panoply [55], and Spons and Shields [51], deploy programs inside enclaves together with a library OS. Such designs decrease the potential impact of the untrusted OS kernel on enclaved software.

cVMs also use a library OS and share design features with these frameworks, but provide effective data sharing that cannot be implemented using enclaves. Enclaves can only share untrusted memory and cannot access each others memory, which is necessary for fast inter-cVM communication. Since enclaves do not trust the host, they must use encryption, impacting performance [50]. Therefore, an interface similar to CP_Files cannot be implemented with enclaves.

Library OSs can be used to de-privilege OS kernel components or create user-level containers. μKontainer [57] offers containers based on the LKL library OS [36]; Williams et al. [68] show that library OSs can be executed efficiently on

top of processes instead of bare VMs; X-Containers [54] offer a cloud platform using library OSs. cVMs share similarities with user-level library OS-based containers but enhance them with strong isolation and a secure communication mechanism using capabilities.

Machine and process isolation. As discussed in §2.1, traditional process-based isolation has shortcomings in terms of performance and TCB size when compared to cVMs. One could envision using virtualization and Intel's `vmfunc` to strike a balance between shared TCB size and communication performance [34]. Virtualization introduces well-known I/O and memory translation overheads, which are costly in a cloud stack, but are not present in cVMs.

8 Conclusions

cVMs are a new VM-like abstraction for cloud applications that use memory capabilities for secure isolation. cVMs include a library OS to minimize how much of the cloud environment is within the TCB. Multiple cVMs safely share an address space, allowing more efficient interaction of application components than when crossing current VM/container boundaries. Their asynchronous read/write and synchronous call interfaces allow capability-unaware, legacy code to run within cVMs.

Acknowledgements. This work was funded by the UK Government's Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme (UKRI grant EP/V000365 "CloudCAP"), and the Technology Innovation Institute (TII) through its Secure Systems Research Center (SSRC). It was also supported by JSPS KAKENHI grant number 18KK0310. We thank our shepherd, Ana Klimovic, and the anonymous reviewers for their helpful comments.

Source code availability. The source code of cVMs, the Intravisor, and various application examples can be found at https://github.com/lsds/intravisor.

References

[1] A self-contained cryptographic library for Python. https://github.com/Legrandin/pycryptodome. Last accessed: June 1, 2022.

[2] Linux containers. https://linuxcontainers.org. Last accessed: June 1, 2022.

[3] CVE-2013-6441. Available from MITRE, CVE-ID CVE-2013-6441, December 2013.

[4] CVE-2014-016021284. Available from MITRE, CVE-ID CVE-2014-0160, December 2014.

[5] CVE-2021-21284. Available from MITRE, CVE-ID CVE-2021-21284, December 2021.

[6] A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2. Last accessed: June 1, 2022.

[7] Alpine Linux. https://alpinelinux.org. Last accessed: June 1, 2022.

[8] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. Last accessed: June 1, 2022.

[9] Arm Morello program. https://developer.arm.com/architectures/cpu-architecture/a-profile/morello. Last accessed: June 1, 2022.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177. ACM, 2003.

[11] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. USENIX Association, April 2008.

[12] Ajay Brahmakshatriya, Piyus Kedia, Derrick P. McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. ConfLLVM: A Compiler for Enforcing Data Confidentiality in Low-Level Code. In *Proceedings of the Fourteenth European Conference on Computer Systems*, EuroSys '19. ACM, 2019.

[13] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, August 2004.

[14] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and R Taheri. Methodology for performance analysis of VMware vSphere under Tier-1 applications. *VMware Technical Journal*, 2(1):19–28, 2013.

[15] Reto Buerki and Adrian-Ken Rueegsegger. Muen - An x86/64 Separation Kernel for High Assurance. *University of Applied Sciences Rapperswil (HSR), Tech. Rep*, 2013.

[16] Building system for CHERI software. https://github.com/CTSRD-CHERI/cheribuild. Last accessed: June 1, 2022, commit a37f5cc.

[17] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Kaladhar Voruganti, and Garth R. Goodson. Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, June 2009. USENIX Association.

[18] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-Based Addressing. *SIGPLAN Not.*, 29(11):319–327, November 1994.

[19] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[20] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy*, pages 56–71, 2016.

[21] CHERI-modified versions of the Flute processor. https://github.com/CTSRD-CHERI/Flute. Last accessed: June 1, 2022.

[22] Darpa FETT Bug Bounty Program. https://fett.darpa.mil. Last accessed: June 1, 2022.

[23] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. *SIGOPS Oper. Syst. Rev.*, 42(2):103–114, March 2008.

[24] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, pages 75–88, January 2006.

[25] FreeBSD adapted for CHERI-MIPS, CHERI-RISC-V, and Arm Morello. https://github.com/CTSRD-CHERI/cheribsd. Last accessed: June 1, 2022.

[26] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 3–18. ACM, 2019.

[27] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. *SIGPLAN Notices*, 52(6):185–200, June 2017.

[28] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael Scott, Kai Shen, and Mike Marty. Janus: Intra-Process Isolation for High-Throughput Data Plane Libraries. Technical report, Technical Report UR CSD/1004, 2018.

[29] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[30] HiFive Unmatched. https://www.sifive.com/boards/hifive-unmatched. Last accessed: June 1, 2022.

[31] Simon P. Johnson. Scaling Towards Confidential Computing. https://systex.ibr.cs.tu-bs.de/systex19/slides/systex19-keynote-simon.pdf, 2019.

[32] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[33] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, July 2019.

[34] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 437–452. ACM, 2017.

[35] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298. USENIX Association, 2017.

[36] Linux Kernel Library. https://github.com/lkl. Last accessed: June 1, 2022.

[37] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64. USENIX Association, November 2016.

[38] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9. 2016.

[39] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. *HASP@ ISCA*, 10, 2013.

[40] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.

[41] Gal Motika and Shlomo Weiss. Virtio network paravirtualization driver: Implementation and performance of a de-facto standard. *Computer Standards & Interfaces*, 34(1):36–47, 2012.

[42] musl libc. https://musl.libc.org. Last accessed: June 1, 2022.

[43] nginx – HTTP and reverse proxy server. https://nginx.org. Last accessed: June 1, 2022.

[44] Fengfeng Ning, Chuliang Weng, and Yuan Luo. Virtualization I/O optimization based on shared memory. In *IEEE Intl. Conference on Big Data*, 2013.

[45] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *arXiv preprint arXiv:1908.11143*, 2019.

[46] Redis is an in-memory database that persists on disk. https://github.com/redis/redis. Last accessed: June 1, 2022.

[47] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jianbo Guan, Jinzhu Kong, Huadong Dai, and Lisong Shao. Shared-Memory Optimizations for Inter-Virtual-Machine Communication. *ACM Computing Surveys*, February 2016.

[48] RISC-V bring-up tracker. https://github.com/carlosedp/riscv-bringup. Last accessed: June 1, 2022.

[49] RISC-V CPU, simple 5-stage in-order pipeline. https://github.com/bluespec/Flute. Last accessed: June 1, 2022.

[50] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. *E*Actors: Fast and Flexible Trusted Computing Using SGX. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, pages 187–200, New York, NY, USA, 2018. ACM.

[51] Vasily A. Sartakov, Daniel O'Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. Spons & Shields: Practical Isolation for Trusted Execution. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2021, pages 186–200, New York, NY, USA, 2021. ACM.

[52] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 575–587. ACM, 2021.

[53] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[54] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 121–135. ACM, 2019.

[55] Shweta Shinde, DL Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[56] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the Fifth European Conference on Computer Systems*, EuroSys '10, pages 209–222. ACM, 2010.

[57] Hajime Tazaki, Akira Moroo, Yohei Kuga, and Ryo Nakamura. How to Design a Library OS for Practical Containers? In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2021, pages 15–28. ACM, 2021.

[58] The Python programming language. https://github.com/python/cpython. Last accessed: June 1, 2022.

[59] There's a RAT in my code: new npm malware with Bladabindi trojan spotted. https://blog.sonatype.com/bladabindi-njrat-rat-in-jdb.js-npm-malware. Last accessed: June 1, 2022.

[60] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238. USENIX Association, August 2019.

[61] Lluís Vilanova, Nadav Amit, and Yoav Etsion. Using SMT to accelerate nested virtualization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 750–761. ACM, 2019.

[62] Carlos Villavieja, Vasileios Karakostas, Lluís Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman Unsal. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, pages 340–349, October 2011.

[63] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2003.

[64] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W Moore, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical report, University of Cambridge, Computer Laboratory, 2019.

[65] Robert NM Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W Moore, Edward Napierala, Peter Sewell, et al. CHERI C/C++ Programming Guide. Technical report, University of Cambridge, Computer Laboratory, 2020.

[66] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[67] Robert NM Watson, Jonathan Woodruff, Michael Roe, Simon W Moore, and Peter G Neumann. Capability hardware enhanced RISC instructions (CHERI): Notes on the Meltdown and Spectre attacks. Technical report, University of Cambridge, Computer Laboratory, 2018.

[68] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as Processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 199–211. ACM, 2018.

[69] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W Filardo, A Theodore Markettos, et al. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Transactions on Computers*, 68(10):1455–1469, 2019.

[70] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[71] Xen project. Event channel internals. https://wiki.xenproject.org/wiki/Event_Channel_Internals. Last accessed: June 1, 2022.

[72] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB. Last accessed: June 1, 2022.

[73] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

KSplit: Automating Device Driver Isolation

Yongzhe Huang*[1], Vikram Narayanan*[2], David Detweiler[2], Kaiming Huang[1], Gang Tan[1], Trent Jaeger[1], and Anton Burtsev[2,3]

[1]Pennsylvania State University
[2]University of California, Irvine
[3]University of Utah

Abstract

Researchers have shown that recent CPU extensions support practical, low-overhead driver isolation to protect kernels from defects and vulnerabilities in device drivers. With performance no longer being the main roadblock, the complexity of isolating device drivers has become the main challenge. Device drivers and kernel extensions are developed in a shared memory environment in which the state shared between the kernel and the driver is mixed in a complex hierarchy of data structures, making it difficult for programmers to ensure that the shared state is synchronized correctly. In this paper, we present KSplit, a new framework for isolating unmodified device drivers in a modern, full-featured kernel. KSplit performs automated analyses on the unmodified source code of the kernel and the driver to: 1) identify the state shared between the kernel and driver and 2) to compute the synchronization requirements for this shared state for efficient isolation. While some kernel idioms present ambiguities that cannot be resolved automatically at present, KSplit classifies most ambiguous pointers and identifies ones requiring manual intervention. We evaluate our solution on nine subsystems in the Linux kernel by applying KSplit to 354 device drivers and validating isolation for 10 drivers. For example, for a complex `ixgbe` driver, KSplit requires only 53 lines of manual changes to 2,476 lines of automatically generated interface specifications and 19 lines of changes to the driver's code. The KSplit analysis of the 354 drivers shows a similar fraction of manual work is expected, showing that KSplit is a practical tool for automating key tasks to enable driver isolation.

1 Introduction

Device drivers have long been and continue to be a major source of defects and vulnerabilities in modern kernels [19,32, 50,65]. Drivers are expected to support a variety of complex protocols and comply with numerous kernel conventions [23, 76,77], creating challenges in ensuring that device drivers operate correctly in the face of concurrent and asynchronous accesses on multiple CPU cores. In addition, while the core kernel is relatively stable, the number of kernel extensions and device drivers is large and continues to grow. A modern Linux 5.12 kernel contains around 8,960 device drivers that account for 67.7% of the kernel source [3], a number that has nearly doubled since 2013. With a rate of 80,000 commits a year, defects and vulnerabilities are an inherent part of the fast growing and evolving driver codebase.

The recent availability of hardware features for efficient isolation [1,5] and system support that leverages such features [40,43,47,61,63,82] have made low-overhead device isolation frameworks practical [66,68]. The upcoming hardware extensions, e.g., native page-granularity support for isolation of kernel code [5], and 16 byte-granularity isolation with memory tagging extensions (MTE) [1], which are key for enabling low-overhead software fault isolation (SFI) implementations [53], will reduce isolation overheads even more.

Despite the availability of low-overhead isolation mechanisms, the task of isolating existing driver code remains challenging. For decades, device drivers and kernel extensions have been developed in the shared memory environment of a monolithic kernel, where they freely exchange references to large and complex data structures (e.g., hierarchical, cyclic data structures with many data and pointer fields) that mix the state of the driver and the kernel. Isolating a driver requires a careful analysis of the flow of execution between isolated subsystems to identify how the complex state of the system is accessed on both sides of the isolation boundary.

Recent techniques to isolate legacy driver code utilize manual analysis of complex kernel-driver dependencies [18,62, 66,68,80], requiring an immense decomposition effort that limits their applicability. Moreover, proposed techniques to automate such analyses [33,72] only address a small fraction of the task. For example, LXFI, an SFI framework, utilized an iterative procedure to identify all the state required for execution of a driver, gradually annotating the missing parts of the shared state [62]. The scale and complexity of modern drivers make such manual analysis impractical. In the Linux kernel, even simple drivers like `msr`, that provide an interface to model specific CPU registers (MSRs), require analysis of 459 functions and around 10,000 object fields that are transitively reachable from the 21 functions of the driver interface. A more complex network driver, `ixgbe`, requires analysis of

*Contributed equally
†Work done partly at the University of California, Irvine

5,782 functions and over 900,000 object fields—a number that is well beyond the reach of manual analysis. Decaf [72] and Microdrivers [33] took initial steps towards automated analysis for driver isolation prior to the advent of efficient isolation hardware, so these works focused on techniques to isolate only a subset of driver functionalty that could be efficiently executed in isolation. As a result, many challenging parts of the drivers, e.g., interrupt handlers and optimized data-plane functions, remained inside the unisolated kernel. In addition, these techniques did not aim to minimize data synchronization overhead and required several manual tasks.

In this paper, we present KSplit, a new framework for isolating device drivers in the Linux kernel. KSplit performs a collection of static analyses on the source code of the kernel and the driver to generate the synchronization code that is required to execute the driver in isolation. Specifically, KSplit identifies the shared state that is accessed by both driver and kernel, computing how this state is used on either side of the isolation boundary, and how it should be synchronized on each kernel-driver invocation, or when a shared synchronization primitive (e.g. a spinlock or an RCU) is invoked. The result of the analysis is a collection of procedure call specifications in the KSplit interface definition language (IDL). The KSplit IDL compiler then generates glue code that ensures synchronization of data structures between isolated subsystems. Some kernel idioms, such as concurrency and complex data structures, present ambiguities that cannot be resolved automatically at present, so KSplit also identifies these specific problems for developers to focus their effort. This allows one to take an existing driver and produce the data synchronization code necessary to run the driver in isolation, automatically, if possible, and identify remaining tasks that require manual intervention, if needed.

Kernel software presents several challenges for developing accurate and scalable analyses for automating the isolation of legacy drivers, which we address in the design of KSplit.[1]

First, modern kernels have evolved to share fine-grained access to large, hierarchical data structures with their drivers, which enables joint, optimized operation over shared state using complex memory references. To compute shared state accurately, KSplit employs a field-sensitive data flow analysis using a modular alias analysis to identify shared fields while accounting for memory references accurately. To compute shared state scalably, KSplit provides a two-stage analysis to identify the kernel functions that could possibly share access to a data structure with a particular driver, enabling more accurate analysis methods to be targeted to the relevant subset of the kernel.

Second, modern drivers execute in a concurrent, fully-reentrant environment of the kernel, which complicates the challenge of ensuring that the shared state is synchronized correctly when the driver is isolated. KSplit provides algorithms to ensure correct synchronization of shared state for driver invocations, nested calls to kernel functions by drivers, and a variety of concurrency primitives, including spin and sequential locks, read-copy-update (RCU), mutexes, and atomics. KSplit provides an analysis to identify concurrency primitives that operate over shared data, finding that such primitives rarely cross the kernel-driver boundary.

Third, kernels utilize a wide range of low-level idioms that create ambiguities for marshaling in synchronization, e.g., sentinel-terminated and sized arrays, tagged and anonymous unions, self-referential data structures like linked lists, etc. To separate complete drivers, these ambiguities need to be resolved automatically. KSplit partitions these data structures into classes to apply algorithms to determine whether marshaling requirements can be inferred or not. KSplit is able to automate most cases and provide warnings for the rest.

We develop KSplit for the Linux kernel and a recent device driver isolation framework, LVDs [68]. KSplit is a fully parallel analysis that takes only a few seconds to complete for simple drivers, and completes within minutes for complex device drivers like `ixgbe`. We evaluate driver isolation using KSplit on 10 Linux device drivers, intentionally choosing drivers representing a wide variety of functionality and kernel programming idioms. Simple device drivers like `msr` can be isolated with no changes to their code, and only 2 lines of IDL changes are required to resolve ambiguities in the driver's IDL. More complex drivers like `ixgbe` require less than 19 lines of driver code changes and only 53 lines of IDL changes for the 2,476 lines of device interface definitions. We also apply KSplit to 354 drivers, finding that the amount of manual effort is expected to be a similar fraction of the driver size. Drivers isolated using KSplit leverage the low-overhead hardware and software isolation mechanisms, retaining 5.4–18.7% of the non-isolated system's performance. Our experience with isolating device drivers confirms that KSplit is a practical tool for enabling isolation of complete, legacy device drivers through the use of emerging low-overhead hardware and software isolation mechanisms.

2 Background: Device Driver Isolation

Over the years, a range of techniques to isolate kernel extensions explored execution of device drivers in clean-slate microkernels [10,12,13,27,30,35,39,44–46,49,57] and virtual machines [14, 15, 31, 34, 55, 70, 75], re-implementing device drivers in safe programming languages [9,11,37,48,56,67,84], developing backward-compatible driver execution frameworks [8,17,22,24,29,36,38,42,52,71,81,83,86], and finally, isolating unmodified driver code with hardware [33,66,68,80] and software [18, 25, 62] mechanisms. While it is possible to enforce isolation of the driver code through programming language safety [11,48,56] and formal verification [7,20], to achieve isolation of unmodified drivers, driver isolation frameworks rely on either hardware isolation mechanisms (e.g., segmentation, paging, extended page table (EPT) switching),

[1] KSplit is developed for Linux, but our techniques can be applied to other commodity kernels.

core-isolation [66], or techniques of software fault isolation (SFI) [18, 25, 62, 85] that enforce a segment-like isolation boundary around the driver code through instrumentation of control flow and memory instructions.

The main challenge in isolating legacy drivers is to provide controlled access to the state that is shared by the kernel and the isolated driver. Commodity operating systems allow kernels to share an address space, and hence, its entire state with drivers, implementing driver operations on objects jointly accessible to both the kernel and the driver. Often these objects have a complex, hierarchical structure, e.g., `sk_buff` network packet buffers, but only a fraction of these objects (i.e., a small subset of their fields) are accessed by both the kernel and the driver in practice, these forming the shared state. In order for the isolated driver to work correctly, KSplit must identify this shared state comprehensively, but to provide efficient isolation, KSplit must not overapproximate the shared state significantly.

Hardware and SFI frameworks take different approaches to protecting access to the state shared between the kernel and the driver. Hardware approaches control access by executing the driver on an isolated copy of the shared state that is synchronized with the kernel on each driver invocation [33, 66, 68, 80]. SFI approaches, in contrast, execute the driver and the kernel on a single copy of the shared state. This eliminates the need for maintaining two copies of the shared state, but requires access-control checks on each memory access to the shared state [62]. To provide fine-grained access control on the kernel state, SFI systems implement a concept of "capability tables", which allow quick byte-granularity lookup of each kernel field accessible to the driver [62].

Irrespective of the isolation mechanism, however, both solutions require analysis of which state can be accessed by the driver and the kernel, and when each access is allowed [62]. Decaf [72] and Microdrivers [33] took a first step in automated analysis of shared state for isolated drivers by computing the state accessed by the driver on each invocation. However, not all of this state is shared with the kernel, as we find that drivers operate on a significant amount of state that is private to the driver. In addition, these projects only decomposed the non-performance-critical driver code into isolated domains to retain reasonable performance.

Historically, isolation in the kernel remained prohibitive due to the high overhead of hardware and software isolation mechanisms. Recent CPUs, however, signal the growing support for low-overhead isolation primitives. Extended page-table switching with VM functions [6] and user-space memory protection keys (MPKs) [6] already provide support for memory isolation with overheads comparable to system calls for Intel machines [43, 63, 68, 82]. The next generation of Intel machines promises to extend MPK with native support for isolation of ring 0 code [5]. Similarly, the newest ARM CPUs provide support for 16 byte-granularity isolation with memory tagging extensions (MTE) [1], which is key for enabling low-overhead SFI implementations [53].

Recent work has shown that using domain-based isolation can be practical. LXDs [66] and LVDs [68] develop a Nooks-like isolation framework using extended page tables to improve boundary-crossing performance, providing an interface definition language (IDL) for specifying which data requires synchronization in driver interfaces. This work demonstrates the potential for the efficient hardware-based protection domain isolation of legacy drivers. However, such isolation required a significant manual effort to develop IDL definitions for complete drivers. While previous work [33, 72] proposed a method to generate the base IDL, configuring the marshalling requirements for a variety of complex data types and handling concurrency was performed manually. While SFI does not require synchronization on boundary crossings, SFI methods must compute essentially the same information to enable correct isolation with good performance.

A variety of projects have explored techniques to automate various aspects of decomposing user-space programs [16, 21, 41, 58–61, 74, 87–89], called *privilege separation* [78], but these techniques fail to address issues critical to isolating kernel code. For example, PtrSplit [59] proposed techniques to compute marshalling requirements for objects based on runtime tracking, but this adds non-trivial overhead. In addition, these techniques are not designed to handle multi-threaded programs like a kernel.

3 KSplit Overview

KSplit transforms complete, shared-memory device drivers into equivalent drivers that can execute in an isolated domain and on an isolated copy of the driver state. Specifically, KSplit identifies the subset of the kernel state that is required for an isolated driver to run, and derives how this state must be synchronized on invocations that cross the isolation boundary, and also at the points where the driver uses concurrency primitives,[2] e.g., atomics, spinlocks, mutexes, ready-copy-update (RCU), etc.

For example, the kernel submits a network packet to a network device with the `ndo_start_xmit()` function:

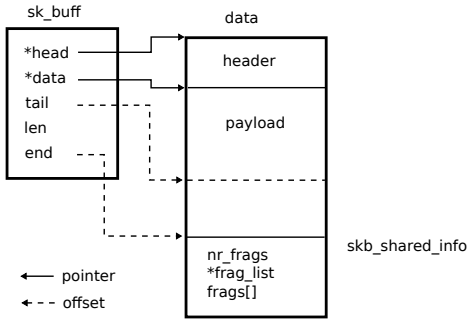
```
1  ndo_start_xmit(struct sk_buff *skb,
2                 struct net_device *netdev)
```

KSplit ensures that all the fields shared between the kernel and driver of all the data structures that are recursively reachable from the two input arguments (i.e., `skb` and `netdev`), and all global kernel variables, are synchronized with the driver. After the invocation completes, the fields updated by the driver are synchronized back to the kernel. Nested invocations into the kernel also trigger synchronization to ensure that the kernel and driver use the current state. If the driver code uses a concurrency primitive that is shared with the kernel, e.g., a global lock, like the `rtnl_lock` used by network device drivers

[2] To distinguish between the synchronization of shared state in general with primitives to synchronize state used in concurrent operations, we refer the latter as *concurrency primitives* in this paper.

```
1  struct sk_buff {
2    struct net_device *dev;
3    unsigned int len, data_len;
4    u8 xmit_more:1;
5    ...
6    sk_buff_data_t tail;
7    sk_buff_data_t end;
8    unsigned char *head, *data;
9    unsigned int truesize;
10 };
```

(a) `sk_buff` definition



(b) `sk_buff` layout in memory

Figure 1: Definition of the `sk_buff` data structure and its layout

to register with the kernel, KSplit synchronizes the state of the driver with the kernel on entry and exit from the atomic region to maintain up-to-date copies in both domains.

KSplit provides software analysis algorithms that 1) compute the subset of the kernel state that is accessed by the driver (i.e., the shared state) and 2) synchronize the shared state on cross-domain invocations and on the use of concurrency primitives that access shared state. While appearing to be conceptually straightforward, isolating legacy drivers is complicated by several factors caused by how drivers are currently deployed in monolithic kernels, specifically:

Complex shared state Kernel data structures often consist of a large number of fields that may be referenced in a variety of ways. The `sk_buff` structure that represents a network packet has 66 fields (5 pointers and 2 offsets) through which 3,132 fields (1,214 pointers) are recursively reachable in other data structures (Figure 1a). The kernel and driver operate jointly on only a small fraction (52 shared fields) of these fields. In addition, like many kernel data structures, the `sk_buff` structure is accessed through complex memory references (Figure 1b). For example, some `sk_buff` pointers are used for in-place access to parts of the network packet, i.e., `head` and `data` mark the beginning of the packet header, and the data regions from which the packet is assembled, respectively.

To compute shared state accurately under these requirements, KSplit employs a field-sensitive data-flow analysis using a modular alias analysis to capture field references common to the kernel and the driver. To do this efficiently, we apply the parameter tree approach [59], which computes aliases intra-procedurally [79] and propagates those alias results inter-procedurally. This approach was employed previously in user-space privilege separation [59]. However, user-space privilege separation aims to isolate sensitive data selected manually by programmers, whereas KSplit needs to identify the data shared between the kernel and a driver automatically. Prior techniques to estimate sharing between the kernel and a driver [33, 72] greatly overestimate shared data because they collect all the fields that the driver will access, instead of those that are actually shared.

Size and complexity of the kernel In order for the isolated driver and kernel to operate correctly, we must identify all the shared state. Using a sound alias analysis, we can over-approximate the shared state, but the kernel is too large (e.g., contains 53,000 functions) to directly apply the field-sensitive analysis needed to compute shared state accurately. KSplit handles this challenge by first performing an analysis to identify the subset of kernel functions that can access the state involved in interaction with the driver. Then, KSplit performs an accurate shared state analysis on this subset of the kernel functions, along with the driver.

Concurrency and parallelism KSplit must ensure that the kernel and the isolated driver operate on up-to-date shared state, regardless of how the kernel and driver interact. The kernel, however, invokes functions of the driver in parallel on multiple CPUs. Moreover, device drivers are concurrent and fully reentrant. As a result, it is possible that the driver updates the shared state that is concurrently accessed by the kernel or vice versa, using one of various concurrency primitives. For example, most drivers use the read-copy-update (RCU) synchronization pattern to synchronize their state across multiple invocations in a lightweight manner, e.g., the `ixgbe` network driver holds an `rcu_read_lock` to access the ring statistics to prevent deallocation of driver queues by a concurrent thread. However, many drivers rely on atomic primitives and critical sections (e.g., `ixgbe` communicates state updates to the New-API (NAPI) state to the softirq framework with atomic variables). Finally, some device subsystems rely on global locks (e.g., `rtnl_lock` in the network subsystem) during driver registration.

KSplit leverages the critical observation that synchronization mechanisms rarely cross the driver-kernel boundary, e.g., out of 73 uses of concurrency primitives in the `ixgbe` driver, only 3 atomic primitives synchronize state across the isolation boundary. We develop a collection of algorithms that carefully classify shared and private critical sections for a range of kernel concurrency primitives (mutexes, spinlocks, sequential locks, atomic primitives, and RCU locks). For shared concurrency primitives, KSplit computes the state that is accessed within the critical section and requires synchronization.

Low-level C idioms Kernel code utilizes a range of low-level idioms that create ambiguities for static analysis (Figure 2). For example, device drivers rely on sentinel values (e.g., `null`) to represent variable-sized arrays, e.g., the PCI subsystem uses the `pci_id_table` array to store a set of devices supported by a particular driver (Figure 2a). To optimize allocation and

```
struct pci_dev { // sized array
  struct resource resource[DEVICE_COUNT_RESOURCE];
};

static const struct pci_device_id ixgbe_pci_tbl[]
  = {
     { PCI_VDEVICE(INTEL, IXGBE_DEV_ID_82598),
       board_82598 },
    { }, /* sentinel */
  };
```

(a) Sized and sentinel arrays

```
#define skb_shinfo(SKB) \
   ((struct skb_shared_info *)(SKB->end))

static inline void
  *blk_mq_rq_to_pdu(struct request *rq)
{
  return rq + 1;
}
```

(b) Collocated data structures

```
ssize_t msr_read(struct file *file,
                 char __user *buf, ...)

dev->bar = ioremap(pci_resource_start(pdev, 0),
                8192);
```

(c) Special memory regions.

```
struct skb_shared_info {
  struct sk_buff *frag_list;
};
```

(d) Recursive data structures

```
union acpi_object {
  acpi_object_type type; /* tag */
  struct {
    acpi_object_type type;
    u64 value;
  } integer;
  ...
};
```

(e) Tagged unions

```
static int ixgbe_set_mac(struct net_device *netdev,
                  void *p) {
  struct sockaddr *addr = p;
  memcpy(netdev->dev_addr, addr->sa_data,
       netdev->addr_len);
  ...
}
```

(f) Opaque pointers

Figure 2: Code idioms typical of the Linux kernel

deallocation of objects, kernel can collocate multiple data structures into one memory area, and use pointer arithmetic to access them (Figure 2b). Further, the lack of a fast array or vector abstraction forces the kernel to use references in place of arrays and keep the length as a separate field. Some memory regions, like user and device I/O memory, require special treatment, when passed into an isolated driver, e.g., marked as allocated in user memory with the `user` attribute (Figure 2c). While recursive data structures are rarely passed across the kernel-driver interface, some drivers use linked lists, and even generic graphs of recursive objects (Figure 2d). Tagged and anonymous unions are used by the driver to implement polymorphic functions that can take generic arguments of a union type (Figure 2e). Device drivers frequently rely on `void*` pointers to express type polymorphism (Figure 2f). Another typical pattern for the kernel APIs is to return an error as a specially formed pointer—this allows a simple unified function signature, e.g., the `struct rquest *blk_mq_alloc_request()` function from the block driver returns a pointer to the block request on successful invocation, but can return a specially formed pointer that represents an error otherwise. KSplit provides support for these cases, and the necessary IDL annotations and library support to generate correct code.

Prior approaches assumed that programmers would provide the annotations to resolve ambiguities in marshaling manually for most cases [33, 51, 62, 66, 68], but that is impractical when isolating complete device drivers. Instead, KSplit takes the opposite approach, aiming to resolve ambiguities in most cases and providing warnings in the remaining ones. For example, `char *` references, such as the `head*` and `tail*` fields in the `sk_buff` data structure, may refer to singletons, arrays, strings, or even other data types (e.g., for type casts). KSplit utilizes a series of classification methods to distinguish among these caonesutomatically, enabling nearly all ambiguities to be resolved for the drivers we have isolated.

Prior work Microdrivers [33], Decaf [72], and FGFT [51] developed static analysis methods aimed at the isolation of legacy driver code. Due to the sheer complexity of the whole-driver analysis, these past approaches were limited to isolating only select driver functions, and supported only a limited subset of kernel idioms. KSplit leverages advances in static analysis: specifically, a combination of an accurate program dependence graph (PDG) representation, and modular alias analysis with parameter trees [59]. This allows KSplit to scale the analysis and implement isolation of the entire driver. A clean separation of shared and private state allows us to scale static analysis and resolve almost all ambiguous annotations required for marshalling of data in the low-level driver code.

3.1 Threat Model and Security Goal

The goal of KSplit is the same as the majority of prior research on driver isolation [35, 66, 68, 80]. Specifically, KSplit aims to improve kernel reliability, i.e., prevent flaws in the driver domain, such as memory errors, from affecting the rest of the kernel. We trust that the kernel domain is free of software flaws, but assume that the driver domain may contain flaws that, for example, may result in writes to kernel memory, possibly causing the kernel to crash.

We leave the feasibility analysis of whether KSplit driver isolation prevents attacks originating from a driver as future work. We note that LXFI [62] prevents certain driver-originated attacks by generating dynamic checks based on user-specified safety conditions at the kernel-driver boundary. However, identifying and specifying safety conditions for individual drivers is a labor-intensive task. Plus a range of security attacks are still possible, such as resource exhaustion (e.g., the driver can allocate objects to consume memory), protocol violations (e.g., the driver can unregister itself from the kernel), and even use-after-free (e.g., driver can trigger deallo-

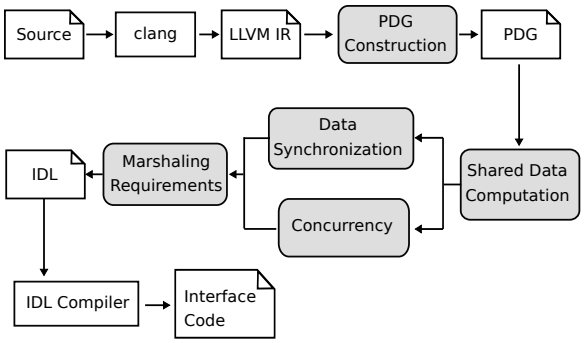


Figure 3: KSplit analysis workflow.

cation of objects in an unexpected way). We, however, believe that KSplit is a critical step towards shaping the foundation of future isolation mechanisms. We plan to study what security guarantees may be possible to achieve automatically as future work. Finally, speculative execution and side-channel attacks are outside the scope of this work as well.

4 KSplit Static Analysis

Figure 3 presents KSplit analysis workflow. KSplit takes the source code (i.e., the code of the kernel and a device driver) as input, and converts it into LLVM IR using Clang, LLVM's frontend. KSplit then provides analyses to: (1) *identify shared data* between the kernel and the driver; (2) *compute data synchronization* on each boundary crossing for that shared data; (3) compute data synchronization for *concurrency primitives* that access shared data; and (4) *infer marshaling requirements* for data types where such requirements are ambiguous, e.g., tagged unions, void pointers, arrays, linked data structures, etc. The result of the analysis is a collection of definitions for the KSplit interface definition language (IDL) compiler. For some cases whose IDL configuration (e.g., size and/or format) remains ambiguous after analysis, KSplit generates warnings for developers to resolve the ambiguity. These warnings must be resolved by developers to obtain a working IDL. The IDL compiler then generates glue code that ensures synchronization of data structures between isolated subsystems.

In this section, we present KSplit's core static-analysis algorithms to address the aforementioned problems. The algorithms are designed to solve these problems in the general case, but the C language is ambiguous about some key information required by the algorithms (e.g., pointer type information). We defer to Section 5 for a discussion of how we leverage C programming idioms used in the Linux kernel to resolve these ambiguities in most cases. While some of these idioms are commonly applied in C programs, some idioms may need to be replaced for other kernels.

4.1 Program Dependence Graph

KSplit reasons about the kernel and drivers using an interprocedural program dependence graph (PDG) [59]. PDG represents individual LLVM instructions as nodes with edges that capture control and data dependencies between instructions. An instruction n_1 is *control dependent* on n_2 if, intuitively, n_2's outcome decides whether n_1 gets executed [26]. An instruction n_1 is *data dependent* on instruction n_2 if n_1 uses some data produced by n_2. Data dependence is critical for determining how the data structures that are exchanged between the driver and the kernel are used in cross-domain invocations. Specifically, KSplit computes how the objects are used by each side of the isolation boundary to then compute data synchronization requirements, as described in Section 4.3. In particular, we need to find all operations that may read or write data, which should be marshaled across the boundary.

Scaling alias analysis with parameter trees A common type of data dependence happens when an instruction writes to a memory region from which another instruction reads. Computing such memory-related data dependenciiiesss requires *alias analysis*, which computes the variables or expressions that may reference (i.e., point to) the same memory object, and are called *aliases*. We must compute aliases in KSplit because we must detect all objects that may be accessed by both the kernel and the drivers. Further, the isolation of the driver code requires an interprocedural alias analysis, as both the kernel and driver code may pass pointers to data objects through function calls, as well as through global variables. The alias analysis problem is known to be undecidable; devising a precise analysis that is scales well and is guaranteed to capture all aliases is a challenge. Current interprocedural alias analysis techniques (e.g., [54,79]), however, do not scale to low-level kernel code with its complex uses of memory references. Instead, we propose to deploy a modular form of alias analysis that enables us to manage scalability more effectively.

In the KSplit approach to modular alias analysis, we employ SVF [79] to compute aliases intra-procedurally and then propagate those alias results inter-procedurally using the parameter tree approach [59]. This allows us to efficiently compute memory dependencies across function boundaries in a context-insensitive way. Specifically, it first constructs PDGs for each function in the program (which includes intra-procedural memory dependencies) and then glues them together by connecting actual parameter trees for arguments at function call sites with formal parameter trees for parameters; details can be found in [59].

To illustrate the idea of parameter trees, consider the `msr_read()` function of the MSR driver. For each argument of the function we construct a parameter tree that represents storage locations that the callee can access. For example, Figure 4 shows a parameter tree for two arguments of the `msr_read()` function: 1) the `file` argument of type `struct file *` and 2) the `int` argument `count`. The parameter tree for the `file` argument has a root node labeled "`file:struct file*`" for representing the storage for the pointer, and a child node labeled "`*file:struct file`" for the memory region that the pointer points to. The references of each storage location in the pro-

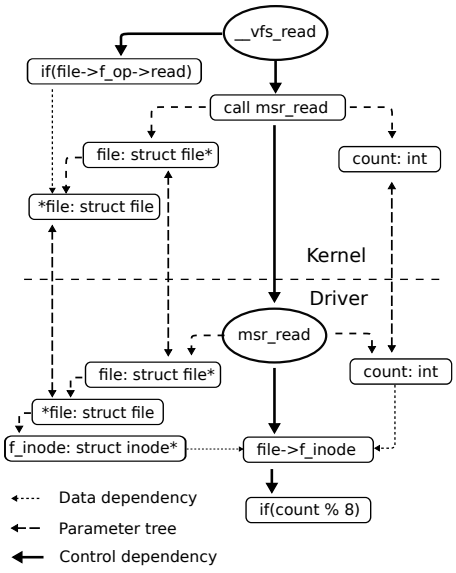


Figure 4: Partial PDG for the `msr_read()` function which is invoked with the `call` instruction from the `__vfs_read()`

gram are connected with corresponding tree nodes through data-dependency edges. We note that, for brevity, the figure does not show the fields of `struct file`; the actual representation maintains information about each field in a separate node to allow field-sensitive analysis.

4.2 Computing Shared and Private Data

Accurate separation of shared and private state is critical for the precision and scalability of KSplit analyses. However, the size of the kernel makes it impractical to perform an accurate analysis to find the shared state at the level of fields (i.e., field-sensitive analysis). On the other hand, the kernel's use of interrupt handlers makes it difficult to ensure that all the code that may impact a particular driver interface invocation has been accounted for. For example, an interrupt handler does not have an explicit caller and is thus unreachable in a typical control-flow graph (CFG) from neither the driver nor regular kernel code. It only runs in response to the corresponding interrupt.

As a result, we develop a shared-state algorithm that first determines the scope of code in the kernel and driver to consider (i.e., the functions and data types that may be shared), as described in steps (1) and (2) of the detailed algorithm below. Then, we perform an accurate, field-sensitive analysis on the PDG. This analysis leverages the modular alias analysis described above to capture the shared state of the kernel and driver in terms of data-structure fields.

The detailed algorithm steps are as follows: (1) the algorithm computes a set of struct types that are accessible by both sides of the isolation boundary. This is performed by collecting all the struct types that are accessible transitively through interface function parameters, global variables, and interrupt handlers. These struct types are referred as *shared struct types*. (2) For each shared struct type, we identify the functions in the kernel and driver that contain variables whose type matches one of the shared struct types. The functions accessible from the isolation boundary in step (1) and those found in step (2) are used to compute the shared state in (3) below. Steps (1) and (2) do not use the CFG and work even for interrupt handlers (unreachable in the CFG). (3) For each set of variables that match a shared struct type, we use the PDG to analyze the accesses via the variables to collect the field accesses for that type. (4) For each field, if the field has accesses from both the kernel and the driver, we consider the field to be shared. Otherwise, the field is private.

The output of the algorithm is a set of shared struct types associated with their shared and private fields. For illustration, the `struct net_device` type contains the following fields (among others): `wanted_features`, `features`, and `hw_features`. By analyzing the `ixgbe` driver and the kernel code, our analysis determines that the `features` field has accesses from both the driver and the kernel, while the other two fields are only accessed in the kernel. Our algorithm determines that `features` is shared, while the other two fields are private to the kernel.

This algorithm relies on two assumptions. First, in step (2), we assume that any state shared between the kernel and driver is accessed using one of the shared struct types from step (1). While this is not guaranteed, the kernel generally obeys typing for the types it shares with the drivers. If we miss a data type, we may under-approximate shared state, causing correctness issues, but we have not found any violations so far. Second, we rely on the observation that the type of a composite object correlates with how it is shared across the isolation boundary. In other words, it is uncommon for one instance of a given type to be shared while a different instance being private; e.g., if a device driver accesses the `inode` field of the `struct file *` object, it is typical that `inode` is shared for all instances of the `struct file *` type. Thus, the analysis cannot determine whether a field of one instance is shared while the same field of another instance is private. The algorithm may over-approximate the shared state, which may cause unnecessary data synchronization, but does not affect correctness.

4.3 Cross-Domain Synchronization

When a function invocation crosses the domain boundary, KSplit synchronizes the shared state required by the callee domain to execute the call. Similarly, when the function returns, the changes the callee made to any shared state must be synchronized back to the caller, to reflect updates on its copy. We develop *parameter access analysis* to compute all data structures and fields that require synchronization.

Basic parameter access analysis At a high level, this algorithm tracks the parameter reads that require data to be synchronized on calls and parameter writes that require data to be synchronized on responses for each cross-domain invocation and any functions reachable from that invocation. Algorithm 1

presents a worklist-based algorithm: 1) for each function in the worklist, it performs an intraprocedural parameter access analysis; 2) it collects call instructions in the function being analyzed and performs an interprocedural analysis; and 3) it repeats steps (1) and (2) until the analysis reaches a fixed point (when the worklist becomes empty). Algorithm 1 computes field usage that is only dependent on parameters passed between domains. Dependence is computed using the parameter-tree alias analysis to ensure an overapproximation.

Algorithm 1: Parameter access analysis

```
Input: G is a PDG, T is a parameter tree, f is the target function of
       a cross-domain call
Output: Access Information Map AM
1  initialize AM to be empty
2  worklist ← {f}
3  while worklist is not empty do
4      f1 ← remove_any(worklist)
5      for node n in T do
6          for instruction i in f1 do
7              if G has a dependence edge from i to n then
8                  AL ← the edge's access label
9                  AM[n] ← AM[n] ∪ AL
10             else if i calls f2 then
11                 worklist ← worklist ∪ {f2}
12             end
13         end
14     end
15 end
```

The analysis goal is to compute a set of *access labels (AL)* for each parameter tree node of a function parameter. The access label of a node represents how the storage represented by the node is used by the callee of a cross-domain call (*READ*/*WRITE*). We further define a global map *AM*, which maps from parameter tree nodes to sets of access labels *AL*. For example, if there is a read access to the `f_inode` field of the `file` data structure, we associate a *READ* label with the parameter tree node that represents the storage of that `f_inode`. After *AM* is computed, the fields for shared state corresponding to nodes with the *READ* label are copied from the caller to the callee when the call happens, and those for shared state with the *WRITE* label are copied from the callee to the caller when the callee returns.

The previous analysis identifies the correct state to synchronize, but might include unnecessary fields because of nested boundary crossings, which cause the call-graph transitive closure to include functions from both sides of the isolation boundary. KSplit distinguishes reads and writes of different domains and avoids sending shared data to a callee if the data is only used in the caller's domain due to a nested call. Similarly, KSplit avoids copying shared data back to the caller if the writes only occur in the caller domain, To do this, KSplit removes shared fields accessed only in the caller domain from the closure computed in Algorithm 1. For the above example, suppose the driver function d reads shared field `fd1` and k' reads shared field `fd2`. The previous analysis determines both `fd1` and `fd2` need to be sent when k calls d. However, our optimization distinguishes the two reads and sends only `fd1`.

4.4 Critical Sections and Atomic Primitives

Modern device drivers are often invoked in parallel on all CPUs of the system, and are fully concurrent outside of small critical sections. The kernel and drivers synchronize access to the shared state through a variety of kernel-provided concurrency mechanisms: atomic operations, spinlocks, sequential and reader/writer locks, read-copy-update critical sections (RCU), etc. To support correct execution of an isolated driver, we provide support for concurrency primitives across the isolation boundary. We identify two large classes of concurrency primitives: locking and lock-free (i.e., atomic operations). For atomic update primitives, e.g., `atomic_inc()`, `atomic_set()`, we perform all updates on the primary copy of the data maintained in the kernel; i.e., drivers call into the kernel to update the primary copy. For synchronization primitives that acquire and release a lock (we support spinlocks, seqlocks, RCU, reader/writer locks, and mutexes), we compute the state that is accessed in each critical section and synchronize it across the isolation boundary. To enforce atomicity across isolated domains, we rely on a mechanism similar to combolocks [33].

The high-level steps for the analysis are as follows: 1) identify *shared* critical sections where cross-subsystem synchronization is required; and 2) identify read/write accesses to shared data in critical sections.

Identifying critical sections To identify critical sections, we perform a search in the CFG of the program, looking for a set of function invocations that implement critical section synchronization primitives, e.g., `spin_lock()`, `mutex_lock()`, etc. For each call to a function marking the beginning of a critical section, we follow the CFG to identify a matching invocation that marks its end, i.e., `spin_unlock()` for `spin_lock()`. Next, we use alias analysis to check whether the beginning (lock) and end (unlock) use the same lock. Finally, we output only critical sections defined by lock/unlock call pairs found by the CFG that are associated with the same lock.

Shared data accesses in critical sections Given a critical section, we identify all shared state that is modified within the critical section. Our goal is to: 1) classify critical sections and atomic operations as either private or shared, i.e., whether the data accessed is private or shared, and then 2) if the critical section operates on shared data, compute the state required for correct synchronization. Specifically, we identify read and write accesses to shared data from inside the critical section (similar to Algorithm 1). For read accesses, we ensure that the state is synchronized right after entering the critical section—this ensures that the code inside operates on a consistent, fresh copy of the state. For write accesses, we synchronize all updates by sending it to the other side of the isolation boundary right before exiting the critical section.

Handling optimized primitives KSplit has support for a variety of concurrency primitives that are optimized to reduce

the use of locking. In most cases, such as sequential locks, the main issue is to determine the corresponding reader and writer critical sections accurately without explicitly locking. For example, we describe how KSplit handles RCU primitives. An RCU lock is often used in manipulating linked list data structures inside the kernel to enable multiple readers and a single writer to access the same data structure concurrently, which reduces the time-consuming lock acquire and release operations. In KSplit, we consider the non-preemptable reader implementation of RCU locks. In this implementation, the start and end of a reader section is defined by calls to `rcu_read_lock()` and `rcu_read_unlock()` functions, respectively. The reader critical section disables preemption. For an RCU writer, KSplit searches for the call sites of functions that may update the data reachable through a pointer used in one of the RCU update primitives, such as `rcu_assign_pointer()` and `rcu_replace_pointer()`. After identifying those reader and writer sections, the same synchronization algorithm as before is used. While this design negates the benefits of RCU locks, they are rarely used across the isolation boundary. Designing a more optimal cross-domain primitive is future work.

5 Low-Level Kernel Programming Idioms

Interface definition language KSplit IDL builds on the ideas from existing driver isolation projects [33, 62, 66]. Specifically, we borrow the idea of "projections", which describe the state synchronized across domains, from LXDs [66] and extend them with rich IDL annotations that provide support for marshaling of low-level C idioms [33]. For every function crossing the boundary of an isolated domain, an IDL remote procedure call declaration is generated.

```
1  rpc netdev_tx_t ixgbe_xmit_frame(
2          projection sk_buff [alloc(callee)] *skb,
3          projection net_device *netdev)
```

For each argument of a composite type, e.g., `struct`, `union`, the IDL includes a projection that lists the shared-state fields that are read or written by the callee function, as determined by the parameter-access analysis (see the example projection for `struct sk_buff` in Section 7.1.1). For ambiguous cases, additional annnotations are included to specify type of the object and in-memory representation (e.g., whether a pointer refers to a singleton or an array, and also type-specific formatting, such as null-termination) and size. KSplit aims to produce these annotations automatically, or generate warnings for programmers to address.

Pointer classification The main challenge for the static analysis is to infer IDL specifications from the low-level type information available in C. For each field type in a projection whose marshaling requirements are ambiguous, we leverage our PDG representation to compute: 1) aliases and def-use chains for references to the ambiguous argument, in order to determine what kinds of operations may be performed on it (e.g., to distinguish singletons and arrays), and 2) all the call sites in which the ambiguous argument is used to infer semantics from uses (e.g., to infer strings from the argument's use in string manipulation functions).

KSplit uses this information to iteratively refine knowledge about the marshaling requirements of arguments, resolving the ambiguities in some cases, and producing specific warnings in others. For example, suppose that an argument has the type `char *`, but we do not know whether this type refers to a singleton, an array, a null-terminated array (i.e., a sentinel array), or another data type altogether (e.g., due to a type cast). KSplit resolves such ambiguities by first leveraging the def-use information of the argument's aliases and then refining the knowledge by applying further analyses. For classification, we employ the CCured method [69], as implemented for LLVM in the NesCheck system [64]. CCured classifies pointers by whether they are involved in type casts (`wild`), are referenced using pointer arithmetic (`sequential`), or neither (`safe`). Pointers classified as safe by CCured/NesCheck are singletons, as these pointers reference only one location. Sequential pointers may be either arrays or structures, although these can be distinguished based on the way they are accessed. Finally, wild pointers involve type cast operations, which make their types ambiguous; although, we can still infer type information in several cases for common patterns.

Once we have performed the classification, we then perform specialized analyses based on the pointer class for further disambiguation:

Sized and null-terminated arrays KSplit can identify arrays whose size is determined at allocation time. It statically detects strings from uses of pointer aliases in any string manipulation functions.

Tagged and anonymous unions Deriving projections for union types is challenging: types and named of the union's fields are lost at the level of LLVM IR, as the compiler treats unions as raw bytes, and simply accesses the fields as offsets. We develop an analysis algorithm that reconstructs field name information by matching the offsets accessed by the IR instructions with the offsets of each field. To marshal the union, the IDL compiler relies on a user-supplied discriminator function to determine the union's active field at runtime.

Recursive data structures KSplit supports marshaling of generic recursive data structures, e.g., linked lists, trees, and graphs. For example, to support a linked list, the static analysis generates a projection that includes a pointer to a projection of the same type as one of the fields. The marshaling code generated by the IDL compiler traverses all the pointers creating a map of visited objects until a fixed point is reached.

Opaque pointers and error pointers If an argument is found to be wild, KSplit can resolve the type in some cases, e.g., void pointers cast to a single type [33]. KSplit handles some other common cases, such as the pattern where kernel APIs may return a reference to either an object or an error.

Other idioms KSplit is able to detect other special cases, such as buffers allocated in user space, co-located data struc-

tures, and "container-of"/"member-of" data structures, to enable special handling (e.g., marshaling of user memory) and targeted warnings (e.g., for marshaling objects collocated within a memory region or a data structure).

6 Implementation

The KSplit system consists of a set of LLVM passes to perform the static analyses, an IDL compiler to generate synchronization code, and a runtime component to track the allocation and deallocation of objects. The LLVM static analysis passes consist of 8,373 SLOC in C++ for PDG construction [59], shared data analysis (Section 4.2), parameter access analysis (Section 4.3), and atomic region analysis (Section 4.4). PDG construction additionally uses the SVF framework [79] for performing the intra-procedural alias analysis. We also use NesCheck [64] for pointer classification required to resolve ambiguities in kernel idioms. To preserve the source semantics, we use optimization level `0` to generate the LLVM bitcode.

We implement KSplit for the LVDs framework, which supports isolation of privileged kernel code through a combination of hardware-assisted virtualization and EPT switching [68]. Specifically, we rely on the LVDs execution environment to run isolated drivers. We, however, implement a new IDL compiler to support synchronization between subsystems; LVDs supported synchronization of only basic types and data structures, but lacked support for arrays, unions, and recursive data structures. The compiler is implemented from scratch in 4,100 lines of C++.

Object lifetimes The main challenge for the runtime is to ensure that object allocation and deallocation on one side of the isolation boundary is reflected on the other side. Tight integration of the kernel and drivers has historically created irregular allocation and deallocation patterns. KSplit relies on a hybrid static and dynamic approach in which the execution runtime tracks new objects and allocates them each time a new object is passed across the isolation boundary. We, however, rely on static analysis to identify deallocation sites and instrument them to propagate deallocations across the isolation boundary.

7 Evaluation

To evaluate KSplit, we utilize CloudLab [73] c220g2 servers configured with two Intel E5-2660 v3 10-core Haswell CPUs running at 2.60 GHz, 160 GB RAM, and a dual-port Intel X520 10Gb NIC. We use an Intel i7-4790K desktop for evaluation of the `alx` network, `xhci` USB host-controller, and Intel ME drivers. Both machines run 64-bit Ubuntu 18.04 Linux, with kernel version 4.8.4.

7.1 Generality of Static Analysis

The main question is whether KSplit can be used as a general tool for the isolation of device drivers in the Linux kernel. To answer this question, we use the KSplit analysis to produce IDL for 354 drivers from multiple Linux subsystems (Table 2), and then evaluate the effectiveness of the analysis and IDL generation algorithms by isolating and validating the correctness of 10 drivers (Table 1). We chose a range of device and protocol drivers that represent typical kernel programming and communication idioms: 1) `msr`: a high-level interface to the model specific registers (MSRs) on the Intel CPUs, which exercises several patterns typical for nearly every Linux device driver—dynamic registration of interfaces and callbacks, synchronization of null-terminated and statically sized arrays; 2) `nullnet`: a software-only network driver that emulates an infinitely fast network adapter, which relies on complex allocation of objects on both sides of the isolation boundary, and implements a fast data plane, requiring careful handling of data structures to achieve optimal performance; 3) `coretemp`: temperature monitoring for CPU cores, which utilizes void pointers and two-dimensional arrays; 4) `sb_edac`: error detection and correction (EDAC) for the Intel Skylake server integrated memory controllers, which requires marshaling of a graph of objects that describe the hierarchy of DRAM banks and memory controllers across the isolation boundary; 5) `null_blk`: a software-only emulation of the NVMe interface; the driver is similar to `nullnet`, i.e., it allows us to stress-test the overheads of isolation on a fast NVMe interface; 6) `ixgbe`: an Intel 82599 10Gbps Ethernet driver, which exhibits several critical characteristics that are interesting for decomposition: first, it relies on atomic operations to update packet statistics in the kernel; second, it exhibits a broad range of asynchronous accesses from system calls, interrupt contexts, software IRQs, and New API (NAPI) threads that implement submission of packets and polling; third, it relies on system timers for several control-plane operations that allow us to test static analysis for support of callback functions dynamically registered with the kernel; 7) `alx`: a Linux Qualcomm Atheros ethernet driver; we select `alx` to compare the complexity and manual effort of decomposing device drivers within the same device class (i.e., we compare three ethernet drivers: `ixgbe`, `alx`, and `nullnet`); 8) `can_raw`: a raw CAN protocol driver using the sockets API, which represents a class of protocol drivers that exhibit typical protocol-layer patterns by interacting with the kernel network stack; 9) `dm_zero` a software block driver that returns `0` on reads and drops writes; `dm_zero` tests if KSplit can fully automate isolation of simple device drivers; 10) `xhci-hcd`: an xHCI protocol driver for supporting USB 3.0, which handles complex interactions of the USB communication protocol.

A wide variety of drivers allows us to examine the generality of the KSplit analyses for producing IDL specifications, and assess the manual effort required for isolation. While we did not run all 354 drivers, we compare metrics related to the effort of isolating an average driver to those we validated. To validate the 10 drivers, we first perform manual tasks required to complete the IDL by resolving all warnings generated by KSplit, and then test the isolated driver trying to evaluate cor-

	coretemp	nullnet	ixgbe	alx	can-raw	sb_edac	null_blk	dm_zero	msr	xhci-hcd
SLOC	562	194	27K	3K	615	2K	690	54	218	10K
Drv.→kern.	21	14	134	61	36	15	36	3	16	45
Kern.→drv.	2	11	81	26	17	1	9	2	5	27
Functions	643	1K	5K	3K	1K	912	1K	133	459	1K

(a) Complexity of driver analysis

Deep copy	31K	46K	999K	214K	153K	24K	75K	11K	24K	134K
Access analysis [33]	127	231	4K	1K	696	91	562	29	66	375
Shared analysis	87	156	3K	831	368	70	406	21	55	265
Boundary analysis	87	155	2K	806	333	70	379	21	51	194

(b) Total number of fields marshaled across all interface functions by each algorithm

Pointers	12K/76	19K/96	404K/1,529	84K/356	60K/178	9K/58	29K/220	4K/16	9K/44	51K/189
Unions	0/0	5/3	114/33	29/17	22/30	0/0	1/12	0/0	0/0	0/7
Critical sections	5/0	5/1	70/3	25/2	19/2	2/0	31/0	0/0	8/0	10/0
RCU	0/0	1/0	8/0	6/0	9/0	0/0	6/0	0/0	0/0	0/0
Seqlock	0/0	0/0	3/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Atomic operations	0/0	25/1	173/35	59/22	49/1	5/0	37/2	3/0	3/0	50/4
Container of	225/4	557/2	2K/20	1K/12	749/8	419/0	627/5	73/3	68/2	1K/6

(c) Impact of shared state optimizations (private/shared)

Singleton	70/0	84/0	1,261/0	307/0	147/0	39/0	183/0	15/0	41/0	172/0
Array	0/1	3/2	92/27	32/2	21/5	5/6	10/5	0/0	0/1	1/0
String	1/0	1/0	2/0	0/0	0/0	2/0	2/0	0/0	1/0	0/0
Wild pointer (void)	2/1	4/0	142/1	12/0	5/0	3/0	17/0	1/0	1/0	16/0
Wild pointer (other)	1/0	0/2	1/3	0/3	0/0	0/3	0/3	0/0	0/0	0/0

(d) Inference type semantics on shared pointers (handled/manual)

Time	17	217	546	190	135	22	490	5	7	238

(e) Analysis execution time (seconds)

Statements	70%	86%	50%	72%	79%	63%	79%	85%	77%	55%
Branches	57%	81%	48%	76%	79%	65%	91%	100%	96%	53%

(f) Test coverage

IDL (LOC)	163	221	2K	674	470	236	306	47	109	1K
IDL changes (LOC)	1	5	53	25	30	5	11	0	2	7
Drv. changes (LOC)	10	6	19	11	12	0	0	0	0	0
False positives	1	25	129	43	30	6	34	2	5	12
Ptr. misclassifications	0	0	7	3	2	2	3	0	2	0
Warnings	1	8	65	22	35	5	20	0	3	7

(g) Manual effort

Table 1: Driver complexity and impact of shared state optimizations.

rectness of isolation what allows us to judge precision and accuracy of the static analysis.

Complexity of driver interfaces To justify the need for automated analysis techniques, we collect several metrics that illustrate the complexity of the 10 drivers isolated using KSplit (Table 1a). The two most complex drivers are `ixgbe` (over 27K SLOC) and `xhci` (over 10K SLOC). The `ixgbe` driver consists of over 2,000 functions, registers 81 callback functions with the kernel, and relies on 134 kernel functions for its operation. Isolation of the `ixgbe` driver involves analysis of the 5,782 functions that may access the state shared between the kernel and the driver. A total of 999,136 fields and scalar arguments are transitively reachable from the arguments of the driver functions that define its isolation boundary (Table 1b). While partial isolation of the `ixgbe` driver was demonstrated before [66, 68], isolation of the complete driver is beyond the reach of manual human analysis.

Impact of shared state optimizations KSplit distinguishes the shared state from the private state, which is critical for the scalability of the analysis algorithms (Section 4.3). We

	char/tty (77)	block (17)	net (89)	edac (13)	hwmon (67)	spi/i2c (38)	usb (53)
SLOC	1047	2535	13302	896	556	471	1340
Drv.→kern.	11	60	25	18	10	14	16
Kern.→drv.	10	16	47	4	5	3	13
Functions	546	2588	2691	839	462	772	784

(a) Complexity of driver interfaces

Pointers	15K /64	53K /310	73K /353	16K /107	10K /61	12K /71	18K /92
Unions	0/2	3/12	7/6	0/2	<1/<1	0/2	<1/4
Crit. sec.	5/<1	51/<1	25/<1	5/<1	6/<1	9/<1	9/<1
Atomic op.	<1/0	6/0	2/0	0/0	<1/0	<1/0	<1/<1
RCU	<1/0	<1/0	<1/0	0/0	<1/0	0/0	<1/<1
Seqlock	9/<1	45/2	45/11	6/0	<1/<1	4/0	10/<1
Container of	145/4	833/3	1K/9	338/2	133/2	207/2	215/3

(b) Impact of shared state optimizations (private/shared)

Singleton	53/0	26/0	303/0	84/0	56/0	66/0	81/0
Array	5/2	27/15	44/20	22/6	2/<1	4/2	4/1
String	<1/0	3/0	<1/0	2/0	<1/0	<1/0	<1/0
Wild (void)	5/<1	18/0	12/1	3/0	1/<1	2/<1	6/<1
Wild (other)	0/<1	0/2	0/3	0/3	0/<1	0/<1	0/2

(c) Inferred type semantics on shared pointers (handled/manual)

Table 2: Performance and complexity metrics across several subsystems (average per driver).

Reference	ixgbe		skx_edac
	nullnet	alx	sb_edac
Shared rpcs	11	73	13
Shared rpcs IDL_{Δ}	+0/-51	+12/-29	+1/-1
Shared rpcs $Annotat._{\Delta}$	0	+3/-3	0
New IDL	77	36	0

Table 3: Similarity within a class

collect the total number of fields in all data structures that are recursively reachable from all the arguments passed across the isolation boundary—previous approaches relied on naive "deep copy" [59] and field-access approaches [33] (Table 1b). Out of 999K fields reachable through the isolation boundary of the `ixgbe` driver, only 4,509 fields are accessed, and an even smaller fraction of them, or 3,029, are shared (Table 1a). Furthermore, by reasoning about nested crossings of the isolation boundary, we reduce this number to 2,669. Most critically, the shared-state optimization radically simplifies the isolation of the driver, as in many cases, complex low-level idioms, e.g., tagged unions, stay on only one side of the isolation boundary (Table 1c). For example, out of 73 critical sections in `ixgbe`, only 3 are shared (`ixgbe` relies on the global `rtnl_lock` to register the driver with the kernel); all RCU and seqlocks are private, and do not trigger cross-boundary synchronization.

Pointer classification To understand how well KSplit supports the classification of pointer references, we characterize the number of supported and problematic pointer patterns in our drivers (Table 1d). In many cases, KSplit is able to infer the types and sizes to enable automatic IDL generation. Table 1d shows that for `ixgbe`, out of 1,529 pointers ("Pointers" in Table 1c) that require marshalling across the isolation boundary, only 31 require manual inspection to generate correct marshaling attributes. There is a small number of misclassified pointers ("Ptr. misclassifications" in Table 1g). We found that these misclassified pointers are sequential pointers that are wrongly classified as singleton pointers; CCured fails to identify pointer-arithmetic operations on them. A detailed study of these misclassified pointers revealed the main reason for misclassification is due to not analyzing library code. For example, the `ixgbe` driver calls the kernel function `pci_request_selected_regions()` with a reference to the driver name string, but the kernel function itself does not perform pointer-arithmetic operations on the reference; instead it passes the reference to a string library. This causes CCured to misclassify the pointer as a singleton pointer. It is possible to resolve some misclassification cases by either extending our analysis to kernel libraries (note, some library functions like `printk()` are challenging for static analysis), and by manually annotating how pointers are used in such functions. For example, if a pointer is passed to a string manipulation function, e.g., `strcmp()`, we can classify the pointer as sequential.

Analysis execution time To understand the practicality of KSplit and its fit for the kernel development toolchain, we measure the execution time of the analysis (Table 1e). The execution time is largely influenced by the number of functions that are involved in the analysis (this number is determined primarily by the size of the driver and by the size of the kernel subsystem the driver interacts with). Complex device drivers that interact with multiple subsystems (e.g., `can_raw`, `null_blk`, `xhci`, and `ixgbe`), require 190-546 seconds to complete. Simple device drivers finish in under a minute.

Precision of the analysis and manual effort To understand the precision of the analysis and the manual effort involved in the isolation of a driver, we compare an automatically-generated IDL with the final, manually-checked and tested IDL used for the isolation of the driver. As we do not have the ground truth, to gain confidence in the correctness of the isolated driver, we execute a collection of tests on each driver. We use Gcov to collect the code-coverage metrics for the tests we run (Table 1f). The code coverage is less than 50% in some cases, since we can only trigger the execution of a subset of the driver code. For example, EDAC drivers support multiple generations of Intel CPUs from Ivy Bridge to Xeon Phi; `ixgbe` supports multiple hardware interfaces, e.g., x540, 82599, 82598; `xhci`, being a protocol driver, has a lot of error handling code, e.g., in a representative function `handle_tx_event()` that handles all the USB transmit events, out of 348 source lines, 198 lines (or 56%) are error handling code that we cannot trigger without fault injection; `sb_edac` driver consists of 1162 lines of code, out of which only 492 (42%) are executable on our Haswell hardware, out of which our tests cover 373 lines of code (thus increasing our coverage

from 63% to 76%).

We collect several metrics that characterize the amount of manual effort involved in resolving IDL warnings (Table 1g). The IDL of a complex driver like `ixgbe`, generated by KSplit, consists of 2,476 lines of code. Isolation of the driver required changing 53 lines of automatically-generated IDL (or 2% of the IDL). We only had to introduce 19 lines of changes to the driver's code, which mostly involve redefinition of certain macros as helper functions (e.g., `setup_timer`, `INIT_WORK`, etc.). KSplit misclassified 7 out of 1,529 pointers shared across the isolation boundary. Two pointers were strings that were passed across the isolation boundary, but were not accessed through pointer arithmetic or string-manipulation functions. One pointer was referring to a DMA memory region that was only accessed by the device but was not involved in any pointer arithmetic in the driver. Four pointers were misclassified due to being passed as arguments to the `memcpy()` function. For smaller drivers, isolation required less than 30 lines of IDL changes. Furthermore, most small drivers required no changes to the driver code.

The "False positives" row lists the number of fields falsely classified by KSplit as shared. We identify them as not shared through manual inspection and driver profiling. The ground truth may be incomplete, so this number represents an upper bound on the number of false positives. The fraction of false positives is generally low (<10%). The dominant reason for false positives are aliases in the shared-data analyses (shared data uses a type-based approach that leads to the overapproximation of fields and `in`/`out` attributes).

Finally, the "Warnings" row shows the number of warnings KSplit's static analyses generate for each driver. These warnings must be resolved by developers to obtain a working IDL.

Similarity within a class A key insight for scaling the isolation to a large fraction of all kernel drivers is grounded on the assumption that drivers within the same class have a significant degree of similarity across their interfaces. Isolation of one driver within the class, therefore, could guide the isolation of other drivers in a relatively straightforward manner, hence amortizing manual effort across the class. To understand the effort involved in isolating multiple drivers in the same class, we choose a base driver within a class and compare it with other drivers in its class (Table 3). We compare two network drivers, `alx` and `nullnet`, to the base `ixgbe` driver. The `alx` driver shares 73 function definitions with `ixgbe` (the total number of functions crossing the isolation boundary in both directions is in Table 1a). After `ixgbe` was decomposed, decomposition of `alx` required changes to 6 annotations and a total 41 lines of changes in the shared part of the IDL.

Generality of IDL generation To judge if KSplit can be used as an isolation tool for the entire population of drivers, we apply it to 354 drivers across nine subsystems in the Linux kernel (Table 2). To make a prediction about the manual effort involved in isolation of the average driver, we collect the same metrics as the ones collected for the validated drivers (Table 1), although all the counts in Table 2 are averages per-driver. In general, we see a huge impact due to the shared-state optimizations (Table 2b) and a low number of problematic pointer instances (i.e., cases that are not "singletons") that could result in warnings (Table 2c). We therefore believe that the effort of isolating an average driver in these subsystems is comparable to the drivers we validated.

	Null	Integer	Array	String	Void	Union
Bytes	0	8	32 * 8	256	4096	24 + 32
Cycles	502	532	690	1310	919	710

Table 4: Overhead of marshaling various data structures

IDL warnings KSplit produces IDL warnings for the following patterns in Table 2c: 1) arrays and "strings" of undetermined size; 2) wild pointers whose type cannot be inferred deterministically from "wild (void)"; 3) anonymous unions in "wild (other)"; and 4) potential cases of collocated data structures in "wild (other)". In general, the number of IDL warnings for each driver is dependent not only on the size of the driver, i.e., lines of code, and complexity of the driver interface, i.e., lines of IDL code, but also on the types of kernel idioms used for communication across the isolation boundary. For example, isolation of the `alx` driver involves an IDL file that consists of 674 lines of code and requires analysis of 22 warnings. The `alx` driver contains 17 anonymous unions, 2 undetermined size arrays and 3 non-void wild pointers. At the same time, isolation of the `can-raw` driver that uses a smaller IDL (470 lines of IDL code) yields 35 warnings. The high number of warnings for `can-raw` is attributed to the 30 instances of anonymous unions and 5 indeterminate-size arrays in its interface.

7.1.1 Case Study: Ixgbe Network Driver

To illustate the process of decomposition, we consider an example, the `ixgbe` driver, that combines a representative set of complex kernel data structures, low-level idioms, and synchronization patterns. As discussed above, separation of shared and private state is critical for reducing the complexity of the IDL required for the isolation of `ixgbe`. KSplit automatically resolves all function pointers that `ixgbe` registers with the kernel as its interface, identifies five `user` and `ioremap` memory regions used by the interfaces of the driver. Out of 143 wild void pointers that `ixgbe` exchanges across the isolation boundary only one required manual intervention ("Wild pointer (void)" in Table 1d). We then had to inspect 3 wild pointers that are type casted between non-void types ("Wild pointer (other)" in Table 1d). The driver requires manual inspection of 27 array pointers (out of 119 exchanged across the boundary). The driver uses one function that returns a pointer-as-error, which is successfully identified by KSplit.

One of the most challenging parts of the `ixgbe` interface is the proper handling of the `sk_buff` data structure, representing a network packet (Figure 1). Several integer fields are used

```
1  projection<struct sk_buff> skb_xmit {
2    projection net_device *dev;
3    unsigned int len;
4    unsigned int data_len;
5    ...
6    void * [alloc_sized<callee>(self->true_size)] head;
7    void * [within<self->head, self->true_size>] data;
8    unsigned int [within<_, self->true_size>] tail;
9    unsigned int [within<_, self->true_size>] end;
10 };
```

Listing 1: Projection of an `sk_buff` data structure

as offsets into the data: 1) `tail` marks the end of the packet's data, and 2) `end` represents the start of the `struct skb_shinfo` that is collocated inside the data memory. The low-level PDG representation of the program allows us to derive that the `skb_shinfo` data structure is allocated within the `data` object. As the `tail` and `end` fields participate in pointer-arithmetic operations, KSplit generates a special `within` IDL attribute that instructs the marshaling code to check that the field is within a specific range, but the range has to be specified manually (Listing 1). KSplit's support for recursive data structures allows us to marshal `sk_buff` buffers that consist of multiple fragments (`sk_buff` contains an optional list of fragments).

7.2 Performance

In general, the performance of the isolated driver is largely determined by the performance of the underlying isolation framework, i.e., LVDs, in our current implementation [68]. We, however, quantify the impact of the KSplit marshaling protocol, and conduct an end-to-end performance measurement of an application using the isolated `ixgbe` driver.

Marshaling overheads We perform microbenchmarks to evaluate the overheads of marshaling various data structures that are commonly used in the Linux kernel (Table 4). For each data structure, the test involves marshaling the data structure, passing it across the isolation boundary, and unmarshaling it. We perform ten million iterations and report an average. On the LVDs system, a null call-reply invocation takes 502 cycles, which includes the overhead of executing the `vmfunc` instruction, saving and restoring general registers, and selecting a stack inside the driver domain. KSplit adds 30 cycles for marshaling simple scalar fields, such as integers. For marshaling tagged unions, we rely on a user-supplied discriminator function that identifies the tag and marshals the union according to the active field's type. In our experiment, we marshal a union that represents a string of 32 characters, which incurs an overhead of 208 cycles.

Memcached To understand end-to-end overheads of isolation on real application workloads, we utilize an experiment that runs memcached, a high-performance, in-memory object-caching system [4] and compare a native, non-isolated kernel with the performance of a system that utilizes an isolated version of the `ixgbe` network driver. We run memcached version 1.5.12 with a single service thread and a cache size of 5GB. We use the memaslap [2] load-generator to send random UDP

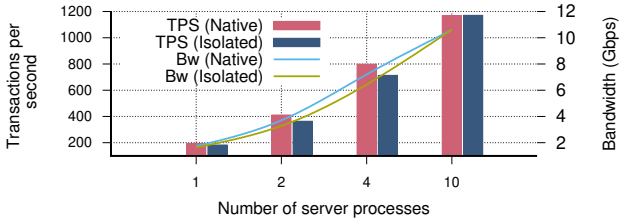


Figure 5: Memcached performance

requests of 64B keys and 1024B values to the server (90% get and 10% set) with a concurrency of 128. To ensure a fair comparison, we limit the number of available cores to 10, as we are limited by the performance of a 10Gbps adapter (all 20 cores would allow isolated drivers to bridge the performance gap, but at a cost of higher CPU utilization). We report both the number of key-value transactions per second and total network bandwidth (Figure 5). For experiments with 1-4 threads, KSplit stays within 5.4-18.7% of the non-isolated system's performance. With 10 threads, both isolated and native drivers saturate the network interface and hence demonstrate nearly identical performance (albeit at higher CPU utilization, due to domain crossings).

8 Conclusions

After decades of research, commodity CPUs are converging on a set of practical hardware mechanisms capable of providing support for low-overhead isolation. With performance no longer being the main roadblock, complexity becomes the main challenge for enabling isolation in commodity systems. Our work on KSplit takes a step forward by enabling isolation of unmodified device drivers in the Linux kernel. A combination of practical static analysis techniques allows us to address the daunting complexity of the driver interfaces—KSplit supports isolation of complex, fully-featured device drivers with only minimal changes or human involvement. While our current implementation works with Linux and a specific isolation framework, we argue that our analysis and state-synchronization techniques are general and can serve as a foundation for a range of isolation solutions enabled by the emerging hardware mechanisms.

Acknowledgments

We thank the ASPLOS'21, OSDI'21, SOSP'21 and OSDI'22 reviewers and our shepherd, Rüdiger Kapitza, for in-depth feedback on earlier versions of the paper. We would like to thank the Utah CloudLab team for continual support in accommodating our hardware requests. Finally, we would like to thank the artifact evaluation committee for numerous comments that greatly improved the artifact. This research is supported in part by the National Science Foundation under Grant Numbers CNS-1527526, OAC-1840197, CNS-1801534, CNS-1816282, and DARPA HR0011-19-C-0106. Vikram Narayanan is partly supported by an IBM PhD fellowship.

References

[1] Armv8.5-A Memory Tagging Extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[2] libmemcached. https://libmemcached.org/libMemcached.html.

[3] LKDDb: Linux Kernel Driver DataBase. https://cateee.net/lkddb/. Accessed on 04.23.2019.

[4] Memcached. https://memcached.org/.

[5] PKS: Add protection keys supervisor (PKS) support. https://lwn.net/Articles/826091/.

[6] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2020. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[7] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, page 175–188, 2016.

[8] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, R Wisniewski, and Jimi Xenidis. Utilizing linux kernel components in K42. Technical report, IBM Watson Research, 2002.

[9] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[10] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 29–44, 2009.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, page 267–283, 1995.

[12] D. W. Boettner and M. T. Alexander. The Michigan Terminal System. *Proceedings of the IEEE*, 63(6):912–918, June 1975.

[13] Bomberger, A.C. and Frantz, A.P. and Frantz, W.S. and Hardy, A.C. and Hardy, N. and Landau, C.R. and Shapiro, J.S. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, 1992.

[14] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *2010 USENIX Annual Technical Conference (USENIX ATC '10)*, 2010.

[15] Bromium. Bromium micro-virtualization, 2010. http://www.bromium.com/misc/BromiumMicrovirtualization.pdf.

[16] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th Usenix Security Symposium*, pages 57–72, 2004.

[17] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[18] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 45–58, 2009.

[19] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[20] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, page 18–37, 2015.

[21] Stephen Chong, Jed Liu, Andrew Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure Web Applications via Automatic Partitioning. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pages 31–44, 2007.

[22] DDEKit and DDE for linux. http://os.inf.tu-dresden.de/ddekit/.

[23] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P:

Safe Asynchronous Event-driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, pages 321–332, 2013.

[24] Kevin Elphinstone and Stefan Götz. Initial evaluation of a user-level device driver framework. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 256–269. Springer, 2004.

[25] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 75–88, 2006.

[26] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[27] Feske, N. and Helmuth, C. Design of the Bastei OS architecture. Technical Report TUD-FI06-07, 2006.

[28] Flux Research Group. CloudLab Web site. http://www.cloudlab.us.

[29] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 38–51, 1997.

[30] Alessandro Forin, David Golub, and Brian N Bershad. An I/O system for Mach 3.0. Carnegie-Mellon University. Department of Computer Science, 1991.

[31] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[32] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP Kernel Crash Analysis. In *Proceedings of the 20th Conference on Large Installation System Administration (LISA '06)*, 2006.

[33] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178, 2008.

[34] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 193–206, 2003.

[35] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 109–114, 2000.

[36] Shantanu Goel and Dan Duchamp. Linux device driver emulation in Mach. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 65–74, 1996.

[37] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC '02)*, page 45–58, 2002.

[38] David B Golub, Guy G Sotomayor, and Freeman L Rawson III. An architecture for device drivers executing as user-level tasks. In *USENIX MACH III Symposium*, pages 153–172, 1993.

[39] Google. Fuchsia project. https://fuchsia.dev/fuchsia-src/getting_started.md.

[40] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pages 401–417, 2020.

[41] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 1016–1031, 2015.

[42] Hermann Härtig, Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An I/O architecture for microkernel-based operating systems. Technical report, TU Dresden, Dresden, Germany, 2003.

[43] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 489–504, 2019.

[44] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[45] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[46] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22. ACM, 2004.

[47] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. Epti: Efficient defence against meltdown attack for unpatched vms. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 255–266, 2018.

[48] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[49] INTEGRITY Real-Time Operating System. http://www.ghs.com/products/rtos/integrity.html.

[50] Trent Jaeger. *Operating System Security*. Morgan & Claypool, 2008.

[51] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 473–484, 2013.

[52] Antti Kantee. *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels*. Doctoral thesis, School of Science, 2012.

[53] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware, 2017.

[54] C. Lattner, A. Lanharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, 2007.

[55] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI '04)*, pages 17–30, 2004.

[56] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 234–251, 2017.

[57] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two Years of Experience with a μ-Kernel Based OS. *ACM SIGOPS Operating Systems Review*, 25(2):51–62, April 1991.

[58] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 285–298, 2017.

[59] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 2359–2371, 2017.

[60] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative Privilege Separation. In *26th ACM Conference on Computer and Communications Security (CCS '19)*, pages 1023–1040, 2019.

[61] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 1607–1619, 2015.

[62] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 115–128, 2011.

[63] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys '19)*, pages 1–15, 2019.

[64] Daniele Midi, Mathias Payer, and Elisa Bertino. Memory Safety for Embedded Devices with NesCheck. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*, page 127–139, 2017.

[65] Brendan Murphy. Automating Software Failure Reporting: We Can Only Fix Those Bugs We Know About. *Queue*, 2(8):42–48, November 2004.

[66] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs : Towards Isolation of Kernel Subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, 2019.

[67] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 21–39, 2020.

[68] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*, page 157–171, 2020.

[69] George Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[70] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 116–132, 2013.

[71] Octavian Purdila. Linux kernel library. https://lwn.net/Articles/662953/.

[72] Matthew J Renzelmann and Michael M Swift. Decaf: Moving Device Drivers to a Modern Language. In *2009 USENIX Annual Technical Conference (USENIX ATC '09)*, 2009.

[73] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[74] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated Partitioning of Android Applications for Trusted Execution Environments. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, pages 923–934, 2016.

[75] Rutkowska, J. and Wojtczuk, R. Qubes OS architecture. *Invisible Things Lab Tech Rep*, 2010.

[76] Leonid Ryzhyk. *On the Construction of Reliable Device Drivers*. PhD thesis, UNSW, January 2010.

[77] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, page 275–288, 2009.

[78] Jerome Saltzer and Michael Schroeder. The protection of information in computer systems. *Proceedings of The IEEE*, 63(9):1278–1308, September 1975.

[79] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266, 2016.

[80] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107, 2002.

[81] Hajime Tazaki. An introduction of library operating system for linux (LibOS). https://lwn.net/Articles/637658/.

[82] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, pages 1221–1238, 2019.

[83] Kevin Thomas Van Maren. The Fluke device driver framework. Master's thesis, The University of Utah, 1999.

[84] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[85] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 203–216, 1993.

[86] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, pages 241–254, 2008.

[87] Yang Liu Yongzheng Wu, Jun Sun and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *International Conference on Automated Software Engineering (ASE)*, pages 323–333, 2013.

[88] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew Myers. Secure program partitioning. *ACM Transactions on Compututer Systems (TOCS)*, 20(3):283–328, 2002.

[89] Lantian Zheng, Stephen Chong, Andrew Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Symposium on Security and Privacy (S&P)*, pages 236–250, 2003.

A Artifact Appendix

Abstract

We release the source code of all software used in this paper along with detailed build instructions and automated scripts used for running the benchmarks as a collection of publicly-hosted Git repositories.

Scope

The artifact allows one to run static analysis on the set of drivers we isolated for this paper and collect metrics that are reported in Table 1, Table 2, and Table 4.

Contents

The artifact consists of the source code for the following subsystems: 1) KSplit analysis framework used to generate interface definition language (IDL) files `https://github.com/ksplit/pdg`; 2) LLVM bitcode files for the drivers analyzed in the paper `https://github.com/ksplit/bc-files` (we provide detailed instructions for how to re-generate the bitcode files, however, to simplify the process of re-creating results reported in the paper, we provide a collection of pre-generated files); 3) KSplit IDL compiler that generates the glue code required to execute the driver in isolation from the IDL files `https://github.com/ksplit/idlc`; 4) a modified Linux kernel that executes isolated drivers in Lightweight Virtualized Domains (LVDs) [68] `https://github.com/ksplit/lvd-linux`; and 5) a modified Bareflank hypervisor that provides secure and efficient isolation boundary based on VMFUNC EPT switching interface used by LVDs `https://github.com/ksplit/bflank`.

Hosting

The artifact is hosted on GitHub. The `README.md` file under `https://github.com/ksplit/ksplit-artifacts` details the steps required to build and run the benchmarks.

We conduct all experiments in the openly-available CloudLab cloud infrastructure testbed [28] and make our experimentation environment available via an open CloudLab [73] profile that automatically instantiates the software setup required to run KSplit: `https://github.com/ksplit/ksplit-cloudlab/`.

Requirements

The KSplit build infrastructure was tested on an x86-64 Ubuntu 18.04 LTS system. The static analysis framework is built and tested against LLVM v10.0.1. We rely on LVDs [68] to execute isolated drivers. LVDs run on any modern Intel x86-64 hardware (Haswell or later) that supports virtualization (Intel VT-x) and EPTP switching via VMFUNC. LVDs rely on a customized Bareflank hypervisor and a modified Linux kernel based on v4.8.4. We have tested KSplit on the following hardware (available in CloudLab): a Cisco UCS C220 machine configured with an Intel Xeon E5-2660 CPU, and a Dell PowerEdge C6420 machine configured with an Intel Xeon Gold 6142 CPU.

Operating System Support for Safe and Efficient Auxiliary Execution

Yuzhuo Jing Peng Huang

Johns Hopkins University

Abstract

Modern applications run various auxiliary tasks. These tasks gain high observability and control by executing in the application address space, but doing so causes safety and performance issues. Running them in a separate process offers strong isolation but poor observability and control.

In this paper, we propose special OS support for auxiliary tasks to address this challenge with an abstraction called *orbit*. An orbit task offers strong isolation. At the same time, it conveniently observes the main program with an automatic state synchronization feature. We implement the abstraction in the Linux kernel. We use orbit to port 7 existing auxiliary tasks and add one new task in 6 large applications. The evaluation shows that the orbit-version tasks have strong isolation with comparable performance of the original unsafe tasks.

1 Introduction

Applications in production frequently require maintenance to examine, optimize, debug, and control their execution. In the past, maintenance was primarily manual work done by administrators. Today, there are increasing needs for applications to self-manage and provide good observability. Indeed, many modern applications execute *auxiliary* tasks. These tasks are designed for various purposes including fault detection [18, 27, 37, 43], performance monitoring [21, 28, 35], online diagnosis [25], resource management [14, 31], *etc.*

For example, PostgreSQL users can enable a periodic maintenance operation called autovacuum [17] that removes dead rows and updates statistics; MySQL provides an option to run a deadlock detection task [30], which tries to detect transaction deadlocks and roll back a transaction to break a detected deadlock; HDFS server includes multiple daemon threads, such as a checkpointer that periodically wakes up to take a checkpoint of the namespace and saves the snapshot.

Essentially, the structure of applications splits into two logical realms of activities (Figure 1)—the *main* and the *auxiliaries*. Despite being peripheral, the latter tasks are important for the reliability and observability of production software.

At the implementation level, though, auxiliary tasks' execution is mixed with the main program's in the same address space, via direct function calls or as threads. Unfortunately, this choice means the auxiliary tasks can incur severe interference to the application's performance, due to unnecessary blocking and contention on CPU, memory, network, and other resources. In addition to costs, bugs in the auxiliary tasks can easily affect the application reliability, *e.g.*, a null-pointer bug inside a checker function can crash the whole process.

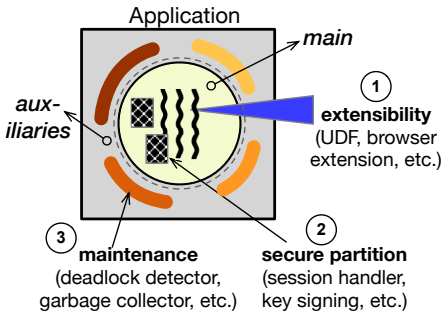


Figure 1: Three protection scenarios for modern applications. This paper focuses on ③.

The alternative is to execute an auxiliary task externally in another process. This choice, however, would impose significant limitations on what can be observed and what can be changed. If the deadlock detector, for example, is run in a separate process, it would not be able to directly inspect the latest transactions or locks; even if it finds a deadlock it could not apply changes to mitigate the issue.

A fundamental problem is that existing OS abstractions for task execution—processes and threads—are designed for the main activities, but are unfit for auxiliary tasks. They force developers to either choose strong isolation but very limited observability and control (in a separate process), or high observability and control but little isolation (in a thread). In this paper, we advocate direct OS support for the trend of *auxiliary execution* to tackle this tension.

OS support for sub-process protection is not new. The systems and security communities have proposed various mechanisms [10, 12, 16, 24, 29, 40, 42, 47]. However, they are designed for two other different purposes. As illustrated in Figure 1, mechanisms such as SFI [42] are designed for *application extensibility* (①). That is, safely execute some *untrusted* third-party extension code, *e.g.*, browser extensions and user-defined-functions in database queries. Another category of abstractions such as Wedge [10] and lwC [24] are designed for *secure partitioning* (②), *i.e.*, protecting some sensitive procedures in the main program, *e.g.*, session handler

or key signing, in case the application is compromised.

These existing mechanisms are insufficient for the third protection scenario (③)—maintenance. The auxiliary tasks are written by the same developers and are trusted. They are also by nature interactive with the main program and need to constantly inspect the latest states of the main program. They often need to additionally alter the main program execution.

In this paper, we investigate this under-explored protection scenario. We summarize the common characteristics of auxiliary tasks, articulate the unique challenges of protecting such tasks, and advocate for special OS support to close this gap.

We then take the first step to propose a new OS abstraction called *orbit* for auxiliary execution. Orbit enables developers to conveniently add a wide range of auxiliary tasks that execute safely and efficiently while assisting the application.

Orbit has several unique features compared to existing sub-process abstractions such as threads, SFI, and lwC. An orbit is a first-class execution entity with a dedicated address space and is schedulable. Each orbit is bound with a main process but provides strong isolation: (i) if an orbit task is buggy and crashes, it does not affect the main process; (ii) orbit executes asynchronously and can be directly enforced with resource control, thus the main process is isolated from an auxiliary task's performance interference. At the same time, orbit provides high observability. Each *orbit*'s address space is mostly a mirror of the main program's. Thus, when the main process calls an orbit, the orbit can run the task functions with the latest main program states. To meet the need for some auxiliary task to change the main process, orbit provides controlled alteration to safely apply updates.

There are two challenges in designing orbit. First, isolation and observability are difficult to achieve together. Second, isolation is known to be costly. Since the main process often calls auxiliary tasks continuously, orbit can incur large performance slowdown to the main process. Optimizations such as using shared memory conflict with the goal of isolation.

To address the first challenge, we design a lightweight memory snapshotting solution that leverages the copy-on-write mechanism and provides *automatic state synchronization* from the main process' address space to orbit's address space whenever the main process calls the orbit task. To address the second challenge, our insight is that while an auxiliary task may inspect various state variables in the main program, the total size of the inspected state *at each invocation* is often a relatively small portion of the entire program state. Thus, we take a simple approach that coalesces only those state variables that an orbit task needs into what we call *orbit areas*. The kernel dynamically identifies the active memory pages in the orbit areas that an orbit invocation requires and only synchronizes these pages to the orbit side.

The lightweight memory snapshotting solution works at page granularity, which has the advantages of simplicity, robustness, and ease of integration with all mainstream OSes without depending on perfect instrumentations as in more complex techniques such as shadow memory. The disadvantage is that the page granularity incurs higher snapshot overhead due to write amplification (snapshot an entire page even if only one small object is changed) and often false sharing (write protection on shared COW pages). We design several optimizations including incremental snapshot, dynamic page mode selection, and delegate objects to reduce the cost.

We have implemented a prototype of orbit in the Linux kernel 5.4.91. To evaluate the generality of the orbit abstractions, we collect 7 auxiliary tasks from 6 large applications including MySQL, Apache, and Redis, and successfully port these tasks using orbit. We also use orbit to write a new auxiliary task for Apache. To demonstrate the isolation capability of orbit, we inject faults to the orbit version of the tasks. Some faults are directly based on real bugs in the task code. The experiments show that the applications are protected from the faults in all cases. We measure the cost of the isolation by comparing the end-to-end application performance. The orbit version applications only incur a median overhead of 3.3%.

In summary, this paper's main contributions are as follows:

- We identify an under-explored category in protection for auxiliary execution and summarize its characteristics.
- We design a new OS abstraction orbit to enable auxiliary tasks that have both strong isolation and high observability.
- We implement orbit in the Linux kernel and evaluate it on real-world auxiliary tasks in large applications.

The source code of orbit is publicly available at:

`https://github.com/OrderLab/orbit`

2 Motivation and Goals

2.1 Auxiliary Tasks

Modern applications often execute various auxiliary tasks designed for assisting reliability, performance, and security. A few typical categories of auxiliary tasks include:

- **Fault detection.** Many applications have checkers to detect faults dynamically. Examples include watchdogs [26] to catch gray failures [19], deadlock checkers, and GC pause detector. Some checkers are instrumented with compilers, such as sanitizers to detect memory leaks.
- **Performance monitor.** It is common for applications to have monitors that collect performance data. For instance, Redis includes a slow log monitor to record queries that take unusually long time.
- **Resource management.** Large applications run resource management routines. For example, Cassandra periodically runs compaction tasks to improve performance for future queries; it also runs a task to asynchronously remove stale records based on past delete requests.
- **Recovery.** Some routines in an application are designed for assisting active recovery. HDFS continuously scans blocks and schedules tasks to reconstruct blocks with low redundancies. Databases also often employ checkpoint threads that flush modified pages and write checkpoint records.

```
const trx_t* check_and_resolve(lock_t* lock, trx_t* trx) {
  do {
    DeadlockChecker checker(trx, lock, mark_counter);
    victim_trx = checker.search();
    if (victim_trx != NULL && victim_trx != trx)
      checker.trx_rollback();
  } while (victim_trx != NULL && victim_trx != trx);
  return victim_trx;
}
```

Figure 2: Deadlock checker function in MySQL.

The workflow of these tasks typically has three steps: (1) read program states; (2) perform inspection work; (3) take actions and modify some states. Depending on their goals, some tasks only read a few program states, while others may inspect lots of states. Some auxiliary tasks are relatively simple that execute synchronously with the main program, *e.g.*, control flow checks [8]. Others are long-running operations that usually execute asynchronously, *e.g.*, in a background thread. Our main focus in this work is the latter type of auxiliary tasks, since they often pose potential issues to the main program.

Note that some existing auxiliary tasks are written in their current forms, *not* because of their inherent nature, but often due to the lack of system support. For example, an existing detection task may execute synchronously, because otherwise the program state may be changed while the task is checking it. However, if an efficient mechanism exists to automatically snapshot the state to be checked, this task could be easily made asynchronous. We aim to provide the support that improves existing auxiliary tasks while enabling novel ones.

2.2 Example: MySQL Deadlock Checker

To make the discussion concrete, we use a representative auxiliary task, the MySQL deadlock checker, as the running example throughout the paper. Figure 2 shows its simplified code snippet. This task is invoked regularly in the main program. Specifically, in handling an update query, MySQL may need to lock a record; if the locking fails, the checking task is invoked. Each checking function invocation takes the blocked lock and the transaction as arguments.

Inside `check_and_resolve`, a deadlock checker instance is created, which runs a search algorithm to inspect the wait-for graph involving the `lock` and `trx` objects as well as other dependent variables. If the checker detects one potential deadlock, it will try to resolve the issue by choosing a victim transaction and rolling it back (modify the state `victim_trx`).

2.3 Safety and Performance Concerns

Developers usually write auxiliary tasks to execute inside the application process. While this choice makes it convenient for the tasks to assist and monitor the main program, their execution poses safety concerns because they execute in the main program's address space. A common issue is a buggy task accessing invalid memory, which crashes the entire application. In other scenarios, a buggy task may cause the main program to get stuck, *e.g.*, a low-priority data gathering thread blocks the high-priority tasks in a similar vein as the infamous Mars Pathfinder incident [36]. Or, the buggy task accidentally modifies some global variables and causes the main program to misbehave. Some issues occur indirectly because of the address space sharing. For example, a defect in HDFS creates too many `SafeModeMonitor` threads and causes the main program to fail with out of memory errors [4].

It might seem that crashing the main program when the auxiliary task is broken is acceptable for some critical auxiliary tasks. For example, since the deadlock detector is important for resolving deadlocks in transactions, if the detector has an invalid memory access, it might be reasonable to crash the main program. However, in practice, crashing the main program is usually too costly (unavailability and slow recovery) and often incurs unintended side effect (inconsistency and data loss), especially considering that the bugs are not from the main program. Alternatively, if we provide strong isolation for auxiliary tasks, we can decouple the fate of the main program from the fates of the auxiliary tasks, which will allow developers to make better choices. For instance, developers can implement a policy that if an auxiliary task dies, it will be automatically restarted and pick up the previous progress, without affecting the main program's execution.

Besides safety, auxiliary tasks can also incur interference to the main program's performance. For instance, we measure the MySQL performance with the deadlock detector task running. The result shows a 3.5%–79.5% drop in the query throughput. This issue was reported by users [1].

In summary, auxiliary tasks are designed to actively improve application reliability and performance, but paradoxically the shared-address-space execution model can cause them to hurt the main program.

2.4 Why Fork or Sandbox Is Insufficient?

To address the safety and performance concerns of auxiliary tasks, two potential alternatives exist: fork and sandbox.

Fork-based Execution Model In this approach, the application makes a `fork()` system call before an auxiliary task executes and switches to run the task functions in the child process. The separate address space provides strong memory isolation. In addition, the task has a copy of address space and thus can inspect any main program states easily. Once `fork()` completes, the main program can continue, while allowing the auxiliary task to execute asynchronously.

Unfortunately, there are several issues. First, the cost is substantial, which includes the creation of a heavy-weight execution entity, as well as the copying of an address space. Even with the copy-on-write optimization, the main program may modify many pages afterward and trigger excessive copying. Moreover, for auxiliary tasks that execute frequently, the fork overhead will be incurred at each task invocation.

Besides overhead, with the auxiliary task running as a child process, it is difficult for the task to perform maintenance work that requires modifying the main program states. For instance, the MySQL checker can identify a victim transaction and

perform a rollback, but the resolution only affects the child process and would not help the parent process.

Sandbox-based Execution Model Another solution is to execute an auxiliary task in a sandbox, which is well-suited to execute untrusted code, *e.g.*, browser renderer. A sandboxed process has reduced privileges in accessing resources including file systems and system calls, and may reside in a separate fault domain using SFI techniques [42].

However, auxiliary tasks are not untrusted codes that sandboxes are designed for. They are written by the application developers and are trusted. Their safety issues arise because of bugs or unintended side effects such as invalid memory access, infinite loops, using too much CPU, etc., rather than accessing unwanted system calls or files. A sandboxed process in a separate fault domain can access only the memory segment allocated to them. It thus gains little observability of the main program and cannot change the main program state.

RPCs or Shared Memory In principle, some aforementioned limitations can be circumvented using RPCs or shared memory. In practice, such workarounds are not favored by developers, because neither model matches with how developers write auxiliary tasks. Developers currently add auxiliary tasks directly in the application codebase and can easily refer to variables in the main program or invoke its functions. To use the RPC model, developers need to convert many variables and functions to be amenable to RPCs. Variables such as `lock` and `trx` in MySQL are difficult to marshal and unmarshal across calls. Frequent RPCs also add large overhead.

The shared memory model similarly requires cumbersome setup and coordination. In addition, the main process would have to wait until the auxiliary task finishes before continuing. Otherwise, the task would inspect inconsistent states. Another issue is that shared memory defeats the isolation purpose. An auxiliary task may need to access variables that scatter across the main program's address space. As a result, the main process may share a large portion of its address space, posing significant safety issues like a thread-based auxiliary task.

3 Orbit: OS Support For Auxiliary Executions

The aforementioned challenges are largely because existing OS abstractions for execution are designed for activities that have clear modularity and isolation boundaries. Auxiliary tasks are inherently interactive with the main program, but it is also desirable to isolate their faults and avoid interference. Developers are forced to choose either an abstraction that offers high observability but weak isolation (*e.g.*, thread), or one with strong isolation but low observability (*e.g.* process).

To address this gap, we propose direct OS support for auxiliary execution with a new abstraction called *orbit*. Orbit offers high observability of another execution entity, while providing strong isolation. Its end goal is to enable developers to create a variety of auxiliary tasks that assist applications in production to enhance the applications' reliability and performance.

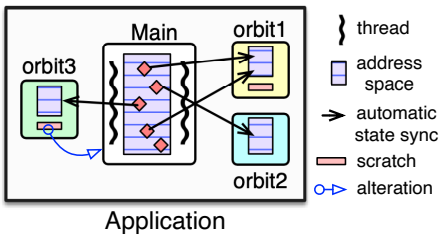


Figure 3: Multiple *orbit*s co-exist with the main program at runtime to provide observability and maintenance support.

3.1 Overview

An orbit task is a lightweight OS execution entity. Each task is bound to "watch" one *target* process. A process can have multiple orbit tasks as shown in Figure 3. They inspect different parts of the target's states for different maintenance purposes. Compared to existing abstractions, *orbit* has several major unique properties:

- **Strong Isolation.** Each orbit task has its own address space. Faults in an orbit would not jeopardize the main program or other orbit tasks. Most orbit tasks execute asynchronously without blocking the main program for a long time.
- **Convenient Programming Model.** The orbit abstraction preserves the current way of how developers write auxiliary tasks. Developers write the orbit task functions within the main program and directly refer to almost any state variables of the main program. They can also easily convert existing functions into orbits. This programming model is close to the thread model that developers are familiar with.
- **Automatic State Synchronization.** A defining characteristic of the orbit task's address space is that it is mostly a mirror of fragments in the target's address space. The fragments are those states that the orbit task needs to inspect. The underlying OS will *automatically* synchronize the specified states to the orbit address space in *one direction*, which occurs before each task invocation in the main program.
- **Controlled Alteration.** A regular orbit only observes the main program, while a privileged orbit is allowed to alter the main program state. However, it cannot change arbitrary state at arbitrary times. The modification has to be made using *scratch space* and well-defined interfaces.
- **First-class Entity.** Orbit tasks are first-class OS entities. They are schedulable like a normal process or thread. This property differs from existing sub-process abstractions such as SFI-based sandboxes and lightweight-context [24], which are subordinates to the main program and *not* schedulable. These abstractions typically have to execute synchronously. An orbit task can be also directly enforced with various limits such as CPU quota.

3.2 Design Challenges and Insight

There are two core challenges that we need to address. First, *how to enable orbit tasks to continuously inspect the main*

API	Description
`orbit *orbit_create(const char *name, orbit_entry entry, void* (*init)(void))`	create an orbit task with a name, an entry function, and an optional initialization function
`int orbit_destroy(orbit *ob)`	destroy the specified orbit task
`orbit_area *orbit_area_create(size_t init_size, orbit *ob)`	create an orbit memory area with an initial size
`void *orbit_alloc(orbit_area *area, size_t size)`	allocate an object of `size` from the orbit area
`long orbit_call(orbit *ob, size_t narea, orbit_area** areas, orbit_entry func_once, void *arg, size_t argsize)`	invokes a synchronous call to the orbit task function with the specific area(s) and arguments, blocks until task finishes
`orbit_future *orbit_call_async(orbit *ob, int flags, size_t narea, orbit_area** areas, orbit_entry func_once, ...)`	invokes an asynchronous call to the orbit task function, returns an `orbit_future` that can be later retrieved
`long pull_orbit(orbit_future *f, orbit_update *update)`	main program waits and retrieves update from orbit future `f`
`long orbit_push(orbit_update *update, orbit_future *f)`	orbit passes update to an existing orbit future `f`

Table 1: Main orbit APIs.

program states conveniently, given that observability and isolation are difficult to achieve together? Second, *how to minimize the performance cost while providing strong isolation?* Isolation inevitably incurs cost. A straightforward design can incur excessive performance slowdowns. Optimizations that can potentially reduce costs, such as using shared memory, are often in conflict with the goal of fault isolation.

Our observations about the characteristics of typical auxiliary tasks reveal insight to address the challenges. While an auxiliary task may inspect various states in an execution, the total size of the inspected state *at each invocation* is often a relatively small portion of the entire program state. In addition, an auxiliary task often performs work incrementally: once the task inspects some state instance in one invocation, the task may not inspect that instance in the next invocation.

4 Orbit Designs

In this section, we describe the designs of the orbit abstraction and how to achieve the properties described in Section 3.

4.1 System Interfaces

The orbit abstraction is exposed through system calls accompanied by a user-level library. Table 1 shows the major APIs.

Developers create an orbit task in place in the application codebase using `orbit_create`, specifying the task entry function. The entry function pointer is defined as `long(*orbit_entry)(void *argbuf, void *store)`, which is similar to the entry function definition in `pthread_create`. However, the orbit entry function executes in a separate address space. This function is also only invoked later by the main program through explicit orbit calls. In other words, the orbit task invocation is decoupled from the orbit creation and can occur *repeatedly*. The `void *argbuf` points to a buffer in the orbit's address space, which is used later during each task invocation to hold the arguments. An optional initialization function can be passed to `orbit_create`. It is useful when some orbit task needs to allocate structure in its address space to keep bookkeeping information. The `orbit_create` returns an orbit handle for the main program to use in later invocations.

```
1  + struct orbit *dlc;
2  + struct orbit_area *area;
3
4  int mysqld_main() {
5  + dlc = orbit_create("dl_checker",check_and_resolve,NULL);
6  + area = orbit_area_create(4096);
7  }
8
9  lock_t* RecLock::lock_alloc(trx_t* trx) {
10   lock_t* lock;
11 - lock = (lock_t*) mem_heap_alloc(heap, sizeof(*lock));
12 + lock = (lock_t*) orbit_alloc(area, sizeof(*lock));
13   return lock;
14 }
15
16 dberr_t lock_rec_lock() {
17   if (status == LOCK_REC_FAIL) {
18 -   check_and_resolve(lock, m_trx);
19 +   dlc_args args = {lock, m_trx};
20 +   orbit_call(dlc, 1, &area, &args, sizeof(dlc_args));
21   }
22 }
```

Figure 4: Using `orbit` to enhance the MySQL deadlock detector. The core logic `check_and_resolve` in Figure 2 remains the same.

The orbit task invocations are done through either the synchronous `orbit_call` or asynchronous `orbit_call_async`. The latter would be particularly common to use. The semantics of the `orbit_call_async` guarantee that the states needed for the task are snapshotted before the API returns. As a result, the main program can continue executing other logic while the orbit task runs concurrently.

This API will return an `orbit_future f`. The main program can wait on `f` later through `orbit_future_get` when it requires knowing the update from the orbit task, just like the typical asynchronous programming models that developers are familiar with. Asynchronous orbit task execution along with the automatic state synchronization feature allows developers to exploit concurrency in the system.

Figure 4 shows an example of using orbit for the MySQL deadlock detector. The task core logic remains the same, but the invocation is split into two steps. Developers use `orbit_create` to create an orbit at the beginning (line 4), which specifies the entry function `check_and_resolve`. An orbit area is created. The allocations of the `lock` (line 12) and `trx` objects are changed to allocate from the orbit area. The original function call (line 19) is replaced with an `orbit_call` to invoke the previously created orbit with the `area` and argu-

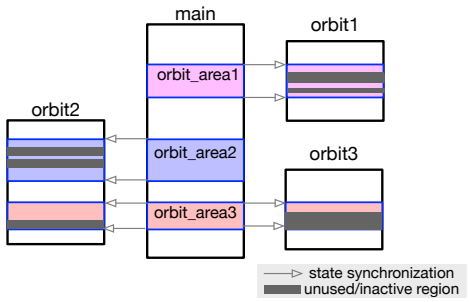


Figure 5: Orbit areas in the main program to be monitored.

ments. Alternatively, developers can use `orbit_call_async` to asynchronously perform deadlock checking.

4.2 Managing Orbit

When a process creates an orbit using `orbit_create`, the kernel internally represents the orbit with a control block and records the target process the orbit is bound with. To avoid intrusive code changes to the Linux kernel function interfaces, we currently re-use the existing `task_struct` (with new fields and a subset of existing fields) to represent the orbit entity.

The main program maintains a `orbit_children` list in its `task_struct`, mapping orbit IDs to the orbit's `task_struct`. Each orbit maintains a `orbit_info` structure in its `task_struct`, that contains the basic execution states of orbit and a FIFO queue of orbit calls.

The kernel also allocates a dedicated address space for the orbit, which is initially kept to a minimum (mostly code pages of the main program). As a first-class OS abstraction, orbit is a schedulable entity and can be enforced with resource limits like a regular process. At the creation time, the orbit is in an *idle* state, waiting for the task invocations. If an orbit task is terminated (*e.g.*, because of its own bugs), it can be configured to be automatically restarted. In that case, after a restart, the orbit task will be reattached to the main program. The main program can explicitly destroy a specific orbit task.

4.3 Synchronizing States to Orbit

Each orbit executes in a separate address space but regularly inspects the state in the main program. To facilitate convenient inspection, the orbit abstraction provides a key feature of automatic synchronization for the referenced state. This automatic synchronization is one-way from the address space of the main to the orbit's. We propose a lightweight memory snapshotting solution for providing this feature.

Determining States One challenge is that an orbit task often inspects state variables that scatter across the main program's address space. Therefore, coarse-grained snapshotting would include too many unneeded objects in the snapshot memory regions, which would not only waste significant memory but also incur large overhead to the application. In addition, while the set of variables an orbit task inspects may be fixed and known at the static compilation time, the dynamic addresses and sizes of these variables can change over time. For example, the MySQL deadlock detector checks different `lock` and `txn` objects in different invocations.

To address this challenge, we take a simple approach that coalesces only those state variables that the orbit tasks need into what we call *orbit areas*. Orbit areas are fragments of the *main program*'s address space. Each orbit area is composed of contiguous virtual pages. An orbit's address space is mostly a mirror of orbit areas (Figure 5). The main program creates an orbit area through `orbit_area_create` with an initial size that is dynamically expandable. This API takes an `orbit` argument. If specified, the kernel will create a memory region in the orbit's address space and ensure it has the same virtual address of the orbit area in the main program before the API returns. Otherwise, this mapping mirroring will be done when an orbit later binds to an orbit area.

For the state variables that may be accessed by some orbit task, their allocation points need to be replaced to allocate from an orbit area through the `orbit_alloc` API. Similarly, these variables can be freed using the `orbit_free` API. The main program can still use these variables like before.

Taking a Snapshot Dynamically, when the main program makes a call to an orbit task function, the kernel identifies the memory pages in the orbit area that contain the variables the orbit task requires. Then the kernel updates the page table entries (PTEs) of these pages to mark them as write protected for copy-on-write (COW). The PTEs are also copied to orbit task's page table with write-protected bit set. For consistent snapshotting, the orbit call will return only after all needed mappings are updated. Afterward, as long as the main program and orbit task do not modify a page, no copying is incurred; otherwise, they will have separate copies of the page. Note that the above snapshotting process occurs on each orbit call, so the mappings in the orbit address space constantly change, but the orbit task is not re-created.

Concurrency To ensure safety under concurrency, the kernel acquires necessary locks (*e.g.*, `mmap_sem` in Linux) while accessing the PTEs in the main program and the orbit. In one orbit call, multiple pages may need to be snapshotted. To provide a consistent snapshot for multi-threaded applications, a conservative solution is to pause all the application threads so that these pages are not modified during the snapshotting. This pausing will incur a significant performance penalty.

We instead rely on application-level synchronization to handle this situation properly. Indeed, if the objects needed in an orbit call may be concurrently modified by some other thread, the application would add proper locks in the original call site to prevent race conditions. For example, the MySQL deadlock checker invocation (Figure 4) is already inside a critical section. Thus, when we port it to an orbit call, the snapshot of the `lock` and `m_trx` objects is consistent.

Locks are intentionally not shared between orbit and the main program, and thus orbit cannot directly alter the main program's lock states. It is possible that a complex orbit task function acquires and releases locks during its execution. In

such cases, acquiring locks can be moved upfront before the orbit call. From our experience of porting tasks that require synchronization (MySQL and Apache), we find that the original auxiliary functions only run within a single global critical section, which makes it straightforward to guarantee consistency. Also, since a consistent snapshot is obtained under a global lock, the orbit task can omit lock acquires in these cases, since it runs singled-threaded in another address space.

Concurrent Orbit Calls Another challenge is to handle state synchronization when some orbit tasks may be invoked concurrently. For example, the MySQL deadlock detector is invoked during request handling. Since MySQL uses multiple threads to handle concurrent requests, the main program may make another orbit call while the previous call is ongoing.

To address this challenge, the kernel maintains a task queue for each orbit (Section 4.4 will describe this part). After introducing the task queue mechanism, we need to ensure `orbit_call(_async)` preserves the semantics that the task invocation will get a consistent snapshot of relevant objects at the time of the API call. The kernel does so by marking COW for the *main program*'s PTEs of relevant orbit area pages, storing the marked PTEs, and returning. The stored PTEs will be installed to the orbit's page table *later* when the queued task executes. This works because, assume that the main program has modified some page in the orbit area while this invocation is in the task queue, COW will be triggered in the main program side and the main program will get a new page. The stored PTEs still point to the old physical page containing the data at the time of the invocation.

Design Choice Rationale Our memory snapshotting leverages the page protection and COW mechanism. Although snapshot at the page granularity can be costly, it integrates well in mainstream OSes and works reliably. Through several optimizations (Section 4.6), we can effectively reduce its performance costs. An alternative solution is to use fine-grained object-level shadow memory, which allocates shadow memory region, uses static analysis to identify and instrument memory writes to the target objects, and checkpoints these writes to the shadow memory region. We did not choose this approach for several reasons. First, the shadow memory consumes significant (often half) of the main program's address space, and because it is in the same address space, the isolation is weak. Second, there can be many objects repeatedly and unnecessarily checkpointed even when the orbit task does not need them. Third, handling concurrency is challenging. Lastly, it makes strong assumptions about the target application and instrumentation accuracies, which are fragile to apply to many complex applications.

4.4 Orbit Task Execution

When an orbit is created, it waits for the main program to make orbit calls. Implementing the task execution is non-trivial, because each call crosses two address spaces. In addition, the orbit may receive different styles of orbit calls, including concurrent calls. The kernel side needs to support these different styles together.

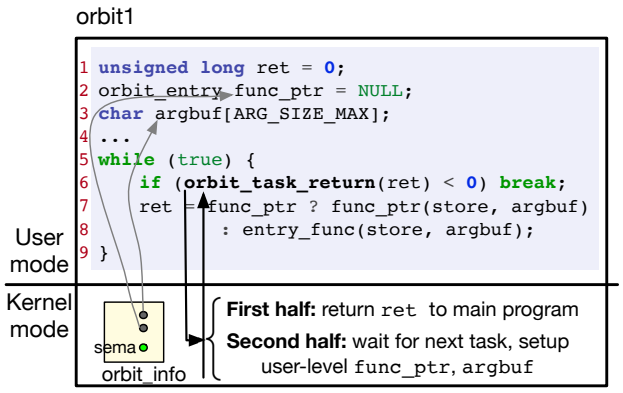


Figure 6: Orbit execution loop waiting for task invocations from main, facilitated by the helper system call `orbit_task_return`.

For supporting potential concurrent calls, the kernel maintains a task queue for each orbit. For each invocation from the main program, the kernel assigns a call id with an internal call struct and inserts it into the queue. The orbit task execution workflow processes the pending invocations in FIFO order. Serializing the task invocation processing makes it much simpler to ensure the correctness of the state synchronization.

To properly implement orbit task execution, we introduce a helper system call `orbit_task_return`. As Figure 6 shows, each orbit is a single-threaded worker executing this loop, and invokes this system call in each iteration. When trapped into the `orbit_task_return` syscall, the kernel knows which main program this orbit corresponds to by looking up the information in its `orbit_info`.

Internally, this kernel function consists of two halves. In the first half, it returns the return value of the *last* orbit call to the main program. Specifically, the kernel stores the passed `ret` value into an internal struct corresponding to the last orbit call, and then *signals* the thread that was executing the last orbit call and blocked waiting for the call to finish. If no orbit call has been made, this first half is skipped.

In the second half, the function waits for the next task from the main program. This is done by waiting on a semaphore in the orbit control block. Once the orbit tasks queue is non-empty, the `orbit_task_return` proceeds and dequeues an invocation. Recall that state snapshotting stores the marked PTEs (Section 4.3) in an array for the pending invocation. The kernel function at this point applies the snapshot by installing the PTEs to the orbit's page table. It then sets up the user-space `argbuf` and `func_ptr`, and returns.

The kernel setups the user-level `argbuf` by copying the orbit call arguments into it. The arguments are typically pointers (*e.g.*, `lock` and `m_trx` in Figure 4), thus only the address values are copied. The actual objects to be referenced in the task are in the orbit area. With the mirroring setup of the orbit area (Section 4.3), the addresses map to equivalent objects. The `func_ptr` is set to either the task entry function or the function pointer specified in the pending `orbit_call`. The

```
void trx_rollback(trx_t *victim) { // within orbit task
 orbit_update *scratch = orbit_update_create();
 orbit_update_add_data(scratch, &victim->version);
 victim->lock.cancel = true;
 orbit_update_add_modify(scratch, &victim->lock.cancel, true);
 orbit_update_add_operation(scratch, pthread_cond_signal,
    &trx->slot->condvar);
 ...
 orbit_push(scratch);
}
void handle_rollback(orbit_future *future) { // in main program
 orbit_update update;
 long ret = pull_orbit(future, &update);
 TrxVersion *version = orbit_update_first(update)->data;
 if (trx_is_alive(version))
  orbit_apply(update);
}
```

Figure 7: Controlled state alteration for MySQL deadlock detector.

latter is particularly useful for an orbit to provide *query* functionalities. For example, if an orbit stores some bookkeeping information, the main program may want to query the orbit about this information occasionally. Finally, the orbit execution loop invokes the appropriate task function with the prepared `argbuf` (line 7 in Figure 6) *at the user level*.

The major task execution workflow described earlier applies to the asynchronous orbit calls as well. The `orbit_call_async` returns an `orbit_future`, which is a reference to the asynchronous task. The main program can later wait on this reference and retrieves updates from the completed asynchronous task, just like the typical asynchronous programming models that developers are familiar with.

4.5 Controlled State Alteration

A privileged orbit is allowed to modify the main program states. One solution is to identify pages in the orbit area that the orbit has modified in its private copies and transparently update the corresponding copies in the main program. The updates are restricted to states belonging to an orbit area. A complication arises if the main program also has since made modifications to some pages in an orbit area. Automatically merging the updates could introduce accidental changes.

To avoid introducing such accidental incorrectness, we instead use a more controlled alteration mechanism by exposing the `pull_orbit` and `orbit_push` system calls. Developers call the `orbit_push` API in the orbit task functions to explicitly decide which updates to push to the main program side. A corresponding call of `pull_orbit` in some main program function will retrieve the updates and explicitly apply the updates to the appropriate state variables. The `orbit_push` API supports pushing flexible data types including raw bytes.

A *scratch* space is backed by some memory region holding the data. The pushing is done efficiently by moving the PTEs of the scratch space pages in the orbit page table to the main program's page table. Besides data, `orbit_push` also supports pushing some operation (function pointer). This is useful if the maintenance operation is difficult to conduct in the orbit side, such as killing some main program's thread.

Example Figure 7 shows an example for the MySQL deadlock detector, which represents a relatively complex use case. Function `trx_rollback` creates a *scratch* `orbit_update` and then pushes a `TrxVersion` by calling `add_data`. This data can later be used to check whether the `victim` transaction is still alive. A following `add_modify` call records the modification of a single field. The next `add_operation` pushes a function with its argument, which will later be invoked in the main program side when the updates are applied and will signal the specified conditional variable. The function pointers are valid for both sides, since the code pages mapping are preserved. The updates are then sent in a batch by calling `orbit_push`.

The `handle_rollback` function then pulls updates from the future. If the task fails, the orbit task is recreated (omitted in the figure). When the main program retrieves an update, it applies the update if the transaction's `version` is still alive.

4.6 Optimizations

We design several optimizations to further reduce the cost of our memory snapshotting. There are two main overhead sources: (1) iterate the PTEs for the active pages in an orbit area, update COW flags, and create mappings in the orbit's address space; (2) page faults when an orbit area is modified.

4.6.1 Incremental Snapshotting

Overhead source (1) is incurred upon each `orbit_call`. In addition, we tear down the orbit's mappings and reset the COW flags of relevant PTEs in the main program when the orbit runs finishes to avoid unnecessary page faults. For orbit areas that have many active pages, this overhead can be significant.

We introduce an incremental snapshotting optimization to reduce this overhead. We keep the mappings after an orbit run finishes. Upon the next `orbit_call`, we iterate through each remained PTE and check if it is the same as the main program's counterpart. If so, we keep it. Otherwise, we recreate the mapping or discard it if the orbit area page is no longer active. Thus, we only pay the mapping cost for the orbit area's pages that are modified by the main program since the last run. One caveat is that keeping the mappings may incur unnecessary page faults. This optimization helps when the main program is not intensively updating the orbit area. We allow developers to pass a flag in an `orbit_call` to indicate whether to enable this mode (keep the mappings).

A second part of this optimization is a region-based marking scheme that aims to reduce the cost of looping through each PTE in an orbit area. We track the PTEs by regions. Specifically, we maintain a bitmap for each range of 512 PTEs (one PMD entry) in the orbit area. A 64-bit bitmap partitions the 512 entries into 64 groups of 8 PTEs. Each bit represents whether the consecutive 8 PTEs have faulted since the last snapshot. During a page fault, the corresponding bit is set to 1. After a snapshot, the snapshotted groups's bits are set to 0. In this way, we can jump to the next group of PTEs that have changed by using bit-wise operation on the bitmap.

```
// allocate with normal malloc
struct trx_t {
  struct {
    ...
-   lock_t* wait_lock;
+   lock_t*& wait_lock;
    ...
  } lock;
+ trx_t_delegate *delegate();
};
```

(a) original full object

```
// allocate with orbit_alloc
struct trx_t_delegate {
  struct {
      lock_t* wait_lock;
  } lock;
};
```

(b) delegate object

Figure 8: Delegate object for the struct `trx_t` in MySQL.

```
// new constructors
trx_t::trx_t(trx_t_delegate *d) : lock(d) {}
trx_lock_t::trx_lock_t(trx_t_delegate *d)
 : wait_lock(d->lock.wait_lock) {}
// creating and binding delegate objects
void trx_init(trx_t *trx) {
 auto delegate = (trx_t_delegate *)orbit_alloc(area,
  sizeof(*trx_delegate));
 new(trx) trx_t(delegate);
}
```

Figure 9: Create and bind delegate object for `trx_t` in MySQL.

4.6.2 Dynamic Page Mode Choice

Overhead source (2) is inherent in the COW mechanism. This cost becomes significant when the orbit area pages are frequently updated by the main program. In this case, COW may perform worse than directly copying the page, which eliminates later page fault penalty to the main program. COW is effective if an orbit area page is infrequently updated.

We support page mode choice (COW or COPY) for an entire orbit area and each page in the orbit area. The former is specified by developers when creating an orbit area. The (likely) update-intensive objects can then be allocated from a COPY-mode orbit area, which will use copying during snapshot. For page-level mode choice, the kernel tracks the statistics of fault rate as `# of faults/# of snapshots` for each page. If the percentage exceeds a heuristic threshold of 30%, we determine the page mode as COPY. Besides, we also impose a limit of 32KB on the total size of COPY pages, and we choose the pages with the highest scores. This is used to prevent exhausting too much memory, and achieve a relatively balanced performance between COPY and COW (because copying large memory region is slower than snapshotting).

4.6.3 Delegate Objects for Large Structs

Complex applications may define large structs, while the states that an orbit is concerned with may be only a small subset of the fields in a large struct. If we allocate the entire large struct from the orbit area, it can incur unnecessary snapshot and page faults due to false sharing.

We use *delegate objects* to mitigate this issue. The basic strategy is to define a delegate struct for the large struct and keep only the fields that are needed in the orbit task functions. Then we allocate the delegate struct from an orbit area but preserve the normal allocation (*e.g.,* `malloc`) for the underlying large struct. Each delegate object has a one-to-one binding to its original struct. It is created at the same time of the original struct as an additional argument to its constructor. To connect the two structs, the relevant fields in the large struct are changed to reference types (*e.g.*, `int` to `int &`, `int *` to `int *&`), and the struct constructor is modified to bind the references to the delegate struct argument. The main program still uses these fields like before without changes.

Figures 8 and 9 show an example of defining and using delegate object for the `trx_t` struct in MySQL. After introducing this delegate object, the main program does not need to change its usages, *e.g.*, `trx->lock.wait_lock` still works. The orbit task function uses the delegate object from `trx->delegate()`.

In our ported systems, we pick those large structs whose total size of accessed fields is smaller than the size of the remaining fields as the target for optimization. Developers can have their own choices to determine what are large structs for delegate object optimization.

4.7 Compiler Support

Our current design requires replacing allocation points for needed state variables (Section 4.3). Some applications already use custom functions to allocate their main objects. In these cases, developers may only need to make minor changes in the custom allocation function to use `orbit_alloc`.

In other cases, developers may need to find individual allocation points and replace them. To help developers with this task, we build an analyzer on top of LLVM [23].

Given an entry function to be converted to an orbit task, *e.g.*, `check_and_resolve` in Figure 2, the analyzer runs forward data-flow analyses to locate all relevant definition and allocation points. Specifically, the analyzer first identifies heap allocation calls in the main program. For each call, it constructs a use graph with the return value variable as the root. Nodes in the use graph include both direct and indirect usage points of the root based on the standard *def-use* chain analysis.

After constructing the use graphs, the analyzer checks whether any use graph can reach the arguments in a callsite of the target function. If so, the allocation point associated with the use graph is included in the result. Besides arguments, the compiler also analyzes the non-local variables referenced in the target function body and leverages the use graphs to identify their allocation points. If no allocation points are found for an argument or non-local variable, the analyzer identifies the definition point (*e.g.*, it is a static global variable) using reaching definition analysis and includes it in the result.

Currently, the analyzer only outputs a list of candidate allocation or definition points. It does not replace these points with `orbit_alloc` automatically, although that is feasible.

5 Evaluation

Our evaluation aims to answer several major questions: (1) *Is orbit general to (re)write auxiliary tasks in complex applications?* (2) *Can orbit-based tasks provide strong isolation?* (3) *How much overhead does orbit incur for achieving isolation?*

No.	Application	Auxiliary Task	Source	Description
t1	MySQL	deadlock detector	port	Automatically detect transaction deadlocks & rollback transaction(s) to break deadlock
t2	Apache	proxy balancer	port	Load balancing to determine suitable proxy backend worker for request
t3	Apache	lock watchdog	new	Periodically check for long mutex lock waits and output notifications to the log
t4	Nginx	WebDAV PUT handler	port	File upload handler for WebDAV PUT requests
t5	Varnish	pool herder	port	Dynamically adjust thread pool sizes
t6	Redis	Slow log	port	A system to log queries that exceeded a specified execution time
t7	Redis	RDB persistence	port	Performs point-in-time snapshots of dataset at specified intervals
t8	LevelDB	background compaction	port	Compact sorted table files to maintain level size limit and improve performance

Table 2: Evaluated auxiliary tasks in six large software.

	small (32 MB)	medium (1G)	large (8G)
orbit	80.51 (8.67)	116.36 (9.12)	115.30 (11.09)
fork	294.24 (27.99)	6859.36 (43.87)	53519.45 (1150.71)

Table 3: Mean latencies (in microseconds) of creating orbit versus process. Numbers in parentheses are standard deviations in 100 runs.

5.1 Evaluation Setup

The experiments are performed in a KVM-enabled QEMU virtual machine with 4-core vCPU and 10GB memory by default, running Debian 10 with our custom kernel. The host machine provides a 20-core Intel Xeon Silver 4114 CPU (2.20GHz), 32GB memory and 480GB SSD running Ubuntu 18.04 LTS. We run all experiments using Linux's default 4KB-sized pages on x86-64, with huge page disabled.

We additionally repeat the experiments on a bare-metal machine, which show matching relative results. Our technical report [20] presents the bare-metal version experiment results.

5.2 Microbenchmark

We first evaluate the performance of creating and invoking orbit with microbenchmarks. We measure the orbit creation under different memory footprint settings of the main program. For a given memory setting, the benchmark program allocates the size, fills it with non-zero data to ensure the kernel actually allocated a physical page for it before running the measured action. It then calls `orbit_create` and measures the latency. We compare the orbit creation with `fork`.

Table 3 shows the result averaged over 100 runs. The initial address space for orbit is minimum with mostly code and stack pages (Section 4.2). Compared to fork, this gives performance benefits for creating isolated address spaces even with a large memory footprint, as most unneeded data are not copied. When the main program has an 8 GB memory footprint, fork is 464× slower than creating an orbit.

We also measure the latency of `orbit_call_async`. Figure 10 shows the result averaged over 20 runs. In general, orbit call time increases almost linearly with the size of orbit area, because it is dominated by the snapshotting cost. For example, making an orbit call with 32MB memory snapshotted takes 272.9 μs, which is comparable to the performance of forking a process with 32MB data shown in Table 3. An orbit call with 8GB snapshotted takes 58.6 ms, which is slightly higher than forking 8GB memory. This is due to the more complicated implementation of snapshotting, such as incremental snapshotting and support for several snapshotting modes.

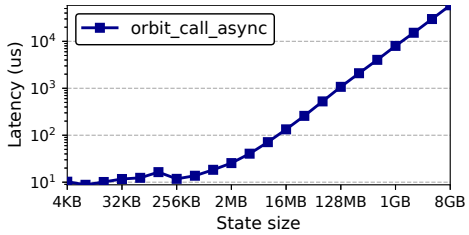


Figure 10: Orbit call latencies with different sizes of snapshot state.

5.3 Applying Orbit on Large Applications

To evaluate the generality of the orbit abstraction, we apply orbit on 6 large applications, MySQL, Apache, Nginx, Varnish, Redis and LevelDB, which have complex codebases and use diverse programming paradigms.

We use orbit to port 7 existing, representative auxiliary tasks in the applications (Table 2). They cover typical auxiliary tasks ranging from fault detection, debugging, resource management, and performance optimization. Two tasks, the Apache proxy balancer and the Nginx WebDAV handler, can be also considered main features. We evaluate them to test the boundaries of tasks that orbit can support. We successfully port all 7 tasks. We run each application's unit tests to verify the ported tasks preserve the original functionalities, even though the tasks now execute the separate address spaces.

We also use orbit to write a new auxiliary task, a lock watchdog, in Apache as an exercise. This task periodically checks if some thread in Apache is stuck and pinpoints the long-holding locks. We add a counter and held locks in thread-local storage. For every lock operation, the main program threads increment the counter, and the number of held locks. A background thread makes an `orbit_call` to the watchdog every second with all threads' counters and held locks. The orbit resets all counters. It also stores historic data of the last held locks and the number of iterations that there is no activity for each thread. When the orbit finds that some thread has has no activity over a threshold (60s), it `orbit_pushes` a return value to inform the main program, which triggers another `orbit_call` to the orbit's diagnosis function that finds the root cause. Figure 11 shows the watchdog thread function.

```
void watchdog_loop() {
 long next_op = WATCHDOG;
 while (true) {
  if (next_op == WATCHDOG)
   next_op = orbit_call(..., wd_areas, wd_func, ...);
  else if (next_op == DIAGNOSIS)
   next_op = orbit_call(..., diag_areas, diag_func, ...);
  ...
 }
}
```

Figure 11: The Apache lock watchdog thread

5.4 Fault Isolation

5.4.1 Fault Injection Testing

We evaluate the isolation capability of orbit by performing fault injection testing on all 8 auxiliary tasks. We inject null pointer deference faults at different times during a task's execution. In all cases, the system successfully isolates the faulty orbit without causing impact to the application and restarts the task gracefully to reattach to the running main process. In some systems, graceful failure handling is implemented by returning an application-specific error code after witnessing an error return code from `orbit_call`. For example, in Apache proxy handler, we return a `HTTP_SERVICE_UNAVAILABLE` after checking the orbit state in main program.

As a first-class OS entity, orbit also provides isolation of performance interference and resource overuse faults in auxiliary tasks. We inject two such faults in Redis slowlog (t6), and mitigate them with `cgroup`. We enforce a memory limit of 256 MB on the orbit task, and inject a memory allocation of 512 MB in orbit task, which this task would never use up. `Cgroup` triggers an OOM kill immediately when the task goes over the memory limit, and the main process gracefully restarts the orbit task. We also inject one CPU hogging for 10 seconds, and modify `cfs_quota` scheduler parameter with `cgroup` to bring CPU usage from taking up one whole core down to 10% of single-core CPU time shown in `top`.

For our newly implemented task in Apache (t3), we inject a long sleep right after one thread has acquired a lock. The watchdog immediately triggers a diagnosis once it finds the counter has not been updated for 60s. The diagnosis function pinpoints the thread ID that holds the lock, along with the location where the lock is acquired.

5.4.2 Real-world Bug Testing

We reproduced 4 *real-world* bug cases from MySQL, Apache, Redis and Nginx that involve the four tasks.

MySQL assertion failure We reproduced the MySQL Bug #28523042 [7]. This bug is introduced in MySQL 8.0 and adds incorrect assertions, which result in assertion failures. We reintroduced this bug into our orbit-enabled MySQL 5.7.31. For demonstration purposes, we modified some part of the expressions that touch the new variables in the 8.0 version, to make the backported code run on the 5.7.31 version.

When a deadlock occurs in the original buggy version, the whole MySQL server crashes, and all clients' connections are dropped. With the orbit-protected deadlock detector, even though the orbit task crashed, the MySQL server is still alive. After the default MySQL lock wait timeout is exceeded, one transaction is chosen as the victim, and all other transactions can continue to finish successfully.

Apache proxy balancer segfault We reproduced Apache Bug #59864 [6]. The user reported that under a proxy balancer configuration with a pair of unavailable fail-over backends pointing to each other, Apache entered infinite recursion when it searched for suitable backend, resulting in stack overflow. We isolate the backend selection in orbit, and successfully catch such failure. Instead of dropping connection, the main program now returns a more meaningful *"Temporary Unavailable"* message when it finds that orbit task has failed.

Furthermore, although web servers like Apache and Nginx often use fault-tolerance mechanisms like multi-process workers, such mechanisms cannot provide fault isolation for concurrent requests within the same worker. When one of the requests triggers a fault, all other connections to this worker also gets disconnected. This applies to both multi-threading (Apache) and event-driven architecture (Nginx) within one worker. Orbit further provides a finer level of isolation by isolating auxiliary tasks within one worker.

Nginx WebDAV segfault Nginx Bug #238 [5] was triggered when a custom WebDAV PUT (*i.e.*, file upload) user request did not include document body. The PUT handler assumes the request body pointer to have been allocated, and thus causes null pointer dereference. Similar to the previous Apache bug, the ported orbit version gracefully catches the failure and returns meaningful messages, while also preventing other requests in the same worker from disruption.

Redis Slowlog memory leak Although Redis uses single-threads for its request processing, its background threads can still cause issues. In case #4323 [2], a race condition happens when both slowlog and asynchronous lazy-free thread decrement a refcount, leading to neither of them freeing the object. Developer mitigated this issue by making a copy of the object. Our orbit implementation, on the other hand, transfers the object from snapshotted orbit area and designates resource management solely to the orbit's address space. Since orbit and the main process do not share the reference counter, race condition is eliminated in the first place.

5.5 Performance Overhead

We measure the end-to-end application performance impact with the orbit-based tasks. We choose application workloads that ensure the auxiliary tasks are triggered frequently.

For MySQL (t1), we run OLTP read-write test provided by the sysbench [3] benchmark tool with 16 clients. We run both Apache watchdog task (t3) and Varnish (t5) using `ab` with 1KB document length and 4 clients. Varnish web cache service uses a stock Nginx as backend. For Apache proxy balancer case (t2), we wrote a custom benchmark using `libcurl` to mix 90% non-proxy requests with 10% proxy requests with 4 clients because `ab` does not support mixed requests. Nginx

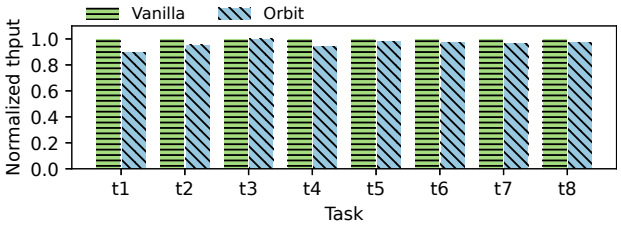


Figure 12: End-to-end application performance with the orbit-based (safe) tasks versus the original (unsafe) tasks.

Task	t1	t2	t3	t4	t5	t6	t7	t8
Calls/s	510.1	1127.8	1	1142.0	1	80.7	0.2	9.9

Table 4: Orbit call frequency in evaluated auxiliary tasks.

WebDAV (t4) benchmark is written in a similar way, with 10% WebDAV upload requests. We run both of the Redis tasks (t6, t7) with YCSB 95% read 5% write test using 32 threads, with either of the tasks enabled separately. We run LevelDB (t8) using a sequential-fill workload with LevelDB built-in benchmark tool to trigger compaction frequently.

Figure 12 shows the normalized throughput for the 8 cases. Most of the (safe) orbit tasks show comparable performance to vanilla (unsafe) tasks. The median overhead is 3.3%. The new task t3 in Apache is compared with the original Apache without our lock watchdog. It has the smallest overhead (0.04%). The largest overhead (10.2%) is the MySQL deadlock checker, which is acceptable considering the strong isolation.

We choose workloads that stress test the orbit tasks. As Table 4 shows, all the tasks are frequently invoked. For example, the MySQL deadlock checker orbit is invoked 510 times per second. In practice, it may not be invoked this frequently. Developers can also add sampling logic for orbit calls.

We also tested less intensive workloads. We reduced the write operations in MySQL (t1)'s OLTP workload, and changed the 90%/10% mix of t2 and t4 to 99%/1% mix. Task t1 and t2 only incur 1.6% and 1.2% overhead, respectively, while t4 has a negligible overhead of 0.18%.

For the MySQL deadlock detector, we implemented a fork version by creating a fork on each invocation to `check_and_resolve`. However, we did not implement IPC to pass results back to the main process, but if implemented, the fork-based performance would become even worse. In comparison, the orbit version has full functionality of pushing updates. We compare the MySQL performance under the three versions of detector: vanilla, fork-based, and orbit-based, using a user workload [1]. Figure 13 shows the result. The orbit version is slower than the vanilla as expected, but 6× faster than the fork-based version. For the orbit version we also compare the performance difference using the synchronous `orbit_call` versus using `orbit_call_async`. Under 8 threads, the performance with asynchronous call is only 1.2% faster than the synchronous call because of limited concurrency opportunities. But under 16 threads, the performance difference becomes much larger as Figure 14 shows.

	Throughput	**Latency**	**Orbit area**	**FPQ**	**TRX size**
No-opt.	1728.0 QPS	340.5 μs	25.7 MB	11.70	912 bytes
Delegate	3308.1 QPS	39.3 μs	1.0 MB	6.91	104 bytes
Changes	+91.4%	-88.5%	-96.1%	-40.9%	-88.6%

Table 5: Optimization effect of delegate object technique. (FPQ stands for page faults per query)

Task	t1	t2	t3	t4	t5	t6	t7	t8
Orbit area	828	20	8	4	8	268	80,644	240
Percentage	0.33	0.40	0.12	0.12	0.001	1.6	76.9	0.65

Table 6: Snapshot sizes (KB) in evaluated auxiliary tasks and their relative percentages (%) of the main program memory footprint.

5.6 Effectiveness of Optimizations

Incremental snapshotting We show the effect of incremental snapshotting by gradually allocating new objects in the orbit area and making orbit calls. We measure orbit call latencies with area sizes from 2 to 256 MB with an increment of 2 MB. Figure 15 shows the result averaged over 20 runs.

Without the optimization, the kernel wastes most cycles walking all the unchanged PTEs and thus requires longer latency. With the optimization, the new data that needs to be snapshotted in every call is a constant (2 MB). For an orbit area of 256 MB, the optimization reduces the latency by 40×.

Delegate Objects We use delegate object technique to minimize states size during snapshots, while also reduce unnecessary page faults due to main process memory writes to the other fields that orbit task does not use.

In the MySQL deadlock detector, we applied delegate object technique to transaction type `trx_t`, lock type `lock_t`, and lock information `lock_sys`. We observe that identifying such optimization opportunities is straightforward. For example, the `trx_t` is 70-field struct with only 4 fields being used in the orbit task, which is clearly an optimization target.

We run the user workload [1] with 16 clients on a 8-core vCPU QEMU VM and compare the throughput, latency, orbit area size, and average page faults per query. Table 5 shows the results. The optimization improves average throughput by 91%, and the orbit call latency to be 7.7× shorter. The total number of page faults throughout the run increases because the throughput also improves, but on average, the number of page faults each request incurs is reduced by 40.9%. In orbit calls, 96.1% of unneeded memory is saved from snapshots. In particular, the delegate object size for `trx_t` is only 11% of the original transaction structure.

5.7 Memory Footprint

Orbit provides efficient snapshotting because orbit only snapshots on necessary data for auxiliary task. We measure the average memory footprint of orbit area that was snapshotted during orbit calls. Table 6 shows the snapshot sizes along with their percentages of the main process's memory footprint. Among the ported tasks, 6 out of 8 allocate less than 1% of process data in orbit area. Redis RDB takes snapshot on its

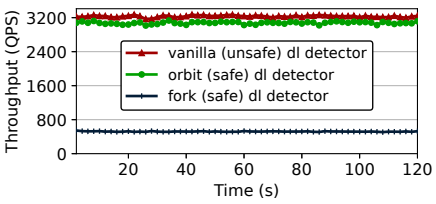


Figure 13: MySQL deadlock detector vanilla versus the orbit-based and fork-based version.

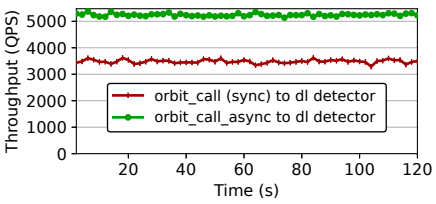


Figure 14: MySQL performance under 16 threads with sync. and async. orbit calls.

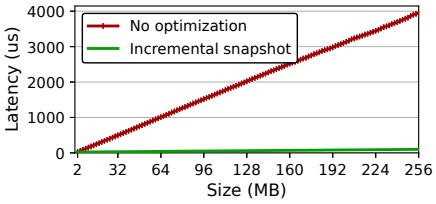


Figure 15: Orbit incremental snapshotting latency on a growing allocation size.

Task	t1	t2	t4	t5	t6	t7	Total
Manual port	7	16	7	3	11	12	56
Compiler	7	56	44	3	20	65	195
Common	5	8	5	1	9	11	39

Table 7: Allocation points in our manual port and compiler result.

key-value dictionary that dominates the memory usage, and thus require the largest portion of memory to be snapshotted.

5.8 Usage Effort

We count the lines of code changes we make to applications in porting the 7 existing auxiliary tasks. The changes include (1) replacing the allocation and free points with orbit allocations; (2) making orbit calls, pushing updates, and applying updates.

The combined changes for (1) range from 40 to 158 lines with a median of 115 lines. The Redis RDB task requires the most changes. We modified some application functions that create certain data structures to provide two versions (one for regular code paths, another for code paths to the orbit task) to avoid putting many unneeded objects in the orbit area. These modifications involved either duplicating the original function or changing its interface. The combined changes for (2) range from 45 to 272 lines with a median of 96 lines.

Our analyzer (Section 4.7) was developed after and motivated by our manual porting effort. We apply it on 6 of the evaluated tasks. The new implementation (t3) case has 0 original allocation points, thus it does not apply. The tool cannot analyze allocations in C++ STL container accurately due to its limited support for STL's complicated internal allocation implementation, thus t8 is excluded.

Table 7 shows the result of manually ported allocation points, detected points and the common ones between the two. From all 56 ported allocation points, our compiler detects 39 of them (70%). The detected points include ported, unported correct points, and false points. For the tasks that have larger number of detected but un-ported points (such as t7), we observe that most of these detected points are correct. They are missed from porting because our workload does not exercise those functionalities. There are also a few cases missing from detection because of unexpected corner cases. For example, a variable in Varnish (t5) used by the auxiliary task was directly allocated on stack instead of using allocator.

6 Discussions and Limitations

As a new abstraction support for auxiliary tasks, our current orbit design has several limitations.

State synchronization Our state synchronization mechanism works at the page granularity, which can incur unnecessary snapshot costs and page faults. Fine-grained object-level snapshotting is feasible but heavily depends on accurate static analysis and instrumentation. We plan to explore potential hybrid solutions that have the advantages of both approaches.

Observable states Our design only considers observing memory states, but not other system states such as file states. Those states would be more complicated to coordinate as they involve kernel and library buffer and position pointer. Creating file snapshots will require a different technique. The tasks we ported are relatively modular and self-contained. For example, our ported checkpointing tasks (Redis RDB, LevelDB compaction) require file operations, but they can create, write, close, and move files within the same orbit context, without the need to share file descriptors with the main program.

Code changes and compiler support We currently require developers to replace the allocation points of needed state variables. For some tasks, a relatively large number of places may need to be replaced. Our future work plans to leverage lightweight memory tracing [32] to dynamically identify the state variables and minimize the code changes.

The analysis in our compiler support for assisting developers to use orbit is basic. Although it supports field-sensitive pointer analysis, it can still miss corner-case allocation points. Developers need to manually find these points. Furthermore, our implementation of *def-use* chain analysis is not accurate enough to determine complex data flow, and thus will yield a handful of false positives. We will enhance the compiler support to enable fully automated porting for developers.

Comparison of programming difficulty Compared to programming with threads, using orbit requires the additional effort to properly change some allocation points. However, although developers do not need to change allocations when using threads, they still need clear knowledge of all the global variables that will be accessed in the thread, and ensure proper synchronizations for them. Thus, developers likely already have some knowledge about the allocation points of these variables. In addition, some of the synchronization would become unnecessary when using orbit. Therefore, the programming overall would be comparable.

Compared to the RPC model, orbit allows developers to write task functions in the same application codebase and directly refer to existing variables and functions. Unlike RPCs

that require code changes to enable object marshalling and unmarshalling, which are difficult for complex objects like transactions and locks, using orbit does not require such changes. With the mirroring orbit area, orbit calls directly access needed objects when crossing the address spaces.

Tolerance of bugs Orbit aims to protect the main program from issues in the auxiliary task execution. It tolerates common bugs such as memory errors in the auxiliary task functions, as well as bugs in the main program that pass bad (or corrupt) values to the auxiliary tasks.

It does not prevent an auxiliary task from sending an incorrect update back to the main program and cause the main program to malfunction. But the orbit abstraction encourages modularization for auxiliary execution, *i.e.*, an orbit task performs most of its operations in a separate address space before pushing updates back. This modularization minimizes the time window for the main program to see bad values and increases the chance that the orbit task itself encounters issues (*e.g.*, dereferencing a bad pointer) before the main program does, which still achieves protection. This is also one reason we choose to provide one-way automatic state synchronization (Section 4.3) with controlled state alteration, instead of a transparent, eager bidirectional state synchronization.

Auxiliary versus main tasks Determining whether a task is auxiliary or main can be subjective. While orbit is designed for auxiliary tasks, it does not require a clear-cut distinction—developers can use it to execute some tasks that they consider as main features for achieving strong isolation. We demonstrate this usage in the evaluation with two cases (t2 and t4).

7 Related Work

There is a wealth of work on protection and fault isolation. They vary widely in their target scenarios (OS extensibility, application extensions, sensitive code, *etc.*), goals (reliability, security, *etc.*), and approaches (software, hardware, hybrid). Our work is complementary to the existing efforts and targets a different, emerging protection scenario—auxiliary tasks in modern applications. Our proposed orbit abstraction aims to provide strong isolation for auxiliary tasks, while also achieving high observability and convenient usage.

SFI [42] is a software isolation technique that restricts the memory accesses of untrusted code in an application by rewriting the application binary. XFI [16] similarly uses binary rewriting to instrument software guards to check memory accesses. Extensive work has followed up this direction, such as NaCl [47] and RLBox [29]. As Section 2.4 elaborates, the sandbox model is not well suited for auxiliary tasks.

Several sub-process OS abstractions [10, 12, 24] provide secure partitioning in applications. They generally use private memory for executing sensitive code to ensure security. Wedge [10] provides the *sthread* primitive to partition an application into compartments and a scheme to tag memory regions and define access rights for the tags. Shreds [12] provides a segment of an execution unit called *shred* and relies on the ARM memory domains hardware feature to provide a private memory pool for each shred. Lightweight context (*lwC*) [24] creates a separate address space for each *lwC* in an application and allows a process to switch to some *lwC* when executing sensitive code. These abstractions typically get executed synchronously and are not independently schedulable.

Determinator OS [9] provides a private workspace model for deterministic parallelism. It runs user code in *spaces* and relies on processes to explicitly synchronize the spaces. Orbit provides automatic, fine-grained state address space synchronization between orbit and the main program. An orbit also has richer features due to its completely different design purpose. SpaceJMP [15] allows a process to define multiple address spaces and switch between address spaces, but with a main goal of enabling applications to use more physical memory rather than fault isolation.

Memory checkpointing takes snapshots of a running program's memory for debugging, failure recovery, quick initialization, *etc.* [11, 13, 22, 46] The checkpoint techniques usually rely on the copy-on-write (COW) mechanism through fork [33, 34, 38] or `mprotect`. On-demand-fork [48] optimizes the fork performance by extending COW to page tables. Orbit synchronizes only needed objects in the orbit areas. Lightweight memory checkpointing [41] uses shadow memory to checkpoint at object granularity. While it is more fine-grained than the page-level COW, shadow memory has several disadvantages for our scenario as described in Section 4.3. Overall, we focus on designing a complete OS abstraction for the isolation of auxiliary tasks. Our work is complementary to existing solutions and can benefit from their optimizations.

Protection schemes are also extensively explored in the context of OS extensibility. To name a few, Nooks [39] provides isolation of device drivers by executing them in different protection domains and using Extension Procedure Call (XPC) for control transfer; Mondrian memory protection (MMP) [44, 45] provides fine-grained protection by using hardware extensions and permission tables.

8 Conclusion

We discuss the trend of auxiliary tasks in applications and the lack of system support for providing safe and efficient execution for these tasks. We propose a new OS abstraction `orbit` to address the gap. Orbit offers high observability and flexible control, while providing strong isolation and efficiency. We evaluate orbit on 8 auxiliary tasks from 6 large applications. The applications achieve enhanced safety with the orbit tasks, and only incur a median of 3.3% performance overhead.

Acknowledgments

We thank the anonymous OSDI reviewers and our shepherd Gerd Zellweger for their valuable feedback. We thank Shreyas Aiyar for his contribution to the orbit compiler. This work was supported in part by NSF grants CNS-1942794, CNS-2149664, CNS-1910133, and CCF-1918757.

References

[1] InnoDB deadlock detection is CPU intensive with many locks on a single row. https://bugs.mysql.com/bug.php?id=49047.

[2] Redis 4.x lazyfree: memory leak may happen when free slowlog entry. https://github.com/redis/redis/issues/4323.

[3] Sysbench. https://github.com/akopytov/sysbench.

[4] Too many safemode monitor threads being created in the standby namenode causing it to fail with out of memory error. https://issues.apache.org/jira/browse/HDFS-5140.

[5] Segfault in DAV module during PUT processing. https://trac.nginx.org/nginx/ticket/238, 2012.

[6] Bug 59864 - segfault when using route-redirect pairs and both servers are disabled/in error mode. https://bz.apache.org/bugzilla/show_bug.cgi?id=59864, 2016.

[7] Bug 28523042 - innodb: assertion failure: lock0lock.cc:7034 in deadlockchecker::search. https://github.com/mysql/mysql-server/commit/97c49a66cea30c96ebc48129f3c4d59ac7a7c913, 2018.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, Alexandria, VA, USA, 2005.

[9] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, Vancouver, BC, Canada, Oct. 2010.

[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, page 309–322, San Francisco, California, 2008.

[11] E. Bugnion, V. Chipounov, and G. Candea. Lightweight snapshots and system-level backtracking. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, page 23, Santa Ana Pueblo, New Mexcio, 2013.

[12] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, 2016.

[13] G. Cox, Z. Yan, A. Bhattacharjee, and V. Ganapathy. Secure, consistent, and high-performance memory snapshotting. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 236–247, Tempe, AZ, USA, 2018.

[14] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, FAST '21, pages 33–49. USENIX Association, Feb. 2021.

[15] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan. SpaceJMP: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 353–368, Atlanta, Georgia, USA, 2016.

[16] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 75–88, Seattle, Washington, 2006.

[17] P. G. D. Group. PostgreSQL's routine vacuuming. https://www.postgresql.org/docs/current/routine-vacuuming.html, 2022.

[18] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[19] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, Whistler, BC, Canada, 2017.

[20] Y. Jing and P. Huang. Operating system support for safe and efficient auxiliary execution (technical report). Technical report, Johns Hopkins University, July 2022. https://orderlab.io/paper/orbit-tr.pdf.

[21] C. H. Kim, J. Rhee, K. H. Lee, X. Zhang, and D. Xu. PerfGuard: Binary-centric application performance monitoring in production environments. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 595–606, Seattle, WA, USA, 2016.

[22] P. F. Klemperer, H. Y. Jeon, B. D. Payne, and J. C. Hoe. High-performance memory snapshotting for real-time, consistent, hypervisor-based monitors. *IEEE Transactions on Dependable and Secure Computing*, 17(3):518–535, 2020.

[23] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Palo Alto, California, 2004.

[24] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-weight contexts: An OS abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, page 49–64, Savannah, GA, USA, 2016.

[25] H. Liu, S. Silvestro, W. Wang, C. Tian, and T. Liu. iReplayer: In-situ and identical record-and-replay for multithreaded applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 344–358, Philadelphia, PA, USA, 2018.

[26] C. Lou, P. Huang, and S. Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS '19, Bertinoro, Italy, May 2019.

[27] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20. USENIX, February 2020.

[28] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393. ACM, 2015.

[29] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium*, USENIX Security '20, pages 699–716, Aug. 2020.

[30] Oracle. MySQL's deadlock detection. https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlock-detection.html, 2022.

[31] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 13–26, Nuremberg, Germany, 2009.

[32] M. Payer, E. Kravina, and T. R. Gross. Lightweight memory tracing. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC '13, pages 115–126, San Jose, CA, USA, June 2013.

[33] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the 1995 USENIX Technical Conference*, USENIX ATC '95, New Orleans, LA, USA, Jan. 1995.

[34] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 235–248, Brighton, United Kingdom, Oct. 2005.

[35] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, page 107–120, Hollywood, CA, USA, 2012.

[36] G. E. Reeves. What really happened on Mars? http://hdl.handle.net/2014/19020, February 1998.

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[38] 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 2004 USENIX Annual Technical Conference*, USENIX ATC '04, Boston, MA, USA, June 2004.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 207–222, Bolton Landing, NY, USA, Oct. 2003.

[40] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security '19, pages 1221–1238, Santa Clara, CA, USA, Aug. 2019.

[41] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum. Lightweight memory checkpointing. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 474–484, Rio de Janeiro, Brazil, June 2015.

[42] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 203–216, Asheville, North Carolina, USA, Dec. 1993.

[43] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 281–294, San Diego, California, 2008.

[44] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 304–316, San Jose, California, 2002.

[45] E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 31–44, Brighton, United Kingdom, 2005.

[46] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2313–2328, Dallas, Texas, USA, 2017.

[47] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 IEEE Symposium on Security and Privacy (SP)*, 2009.

[48] K. Zhao, S. Gong, and P. Fonseca. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 540–555, Online Event, United Kingdom, 2021.

From Dynamic Loading to Extensible Transformation: An Infrastructure for Dynamic Library Transformation

Yuxin Ren, Kang Zhou, Jianhai Luan, Yunfeng Ye,
Shiyuan Hu, Xu Wu, Wenqin Zheng, Wenfeng Zhang, Xinwei Hu
Poincare lab, Huawei Technologies Co., Ltd, China

Abstract

The dynamic linker and loader has been one of the fundamental software, and more than 99% of binaries are dynamically linked on Ubuntu. On one hand, vendors are going to break production software into more and more dynamic libraries to lower the maintenance cost. On the other hand, customers require the dynamic loader to provide rich functionalities to serve their isolation, security, and performance demands. However, existing dynamic loaders are implemented in a monolithic fashion, so they are difficult to extend, configure and optimize.

This paper presents iFed, an infrastructure for extensible and flexible dynamic library transformation. We design iFed in a pass-based architecture to compose various functional and optimization passes. iFed uses a runnable in-memory format to represent libraries and coordinate among multiple transformation passes. We further implement two optimization passes in iFed, which efficiently leverages hugepages and eliminates relocation overhead. iFed is implemented as a drop-in replacement of the current system default dynamic loader. We evaluate iFed and its optimization passes with a wide range of applications on different hardware platforms. Compared to the default `glibc` dynamic loader, iFed reduces an order of magnitude of TLB miss. We improve the throughput of a dynamic website by 13.3%, along with a 12.5% reduction of tail latency without any modifications to the applications.

1 Introduction

Since the 1990s, dynamic linkers and loaders have been one of the most critical software tools for computer programs and applications [11, 15, 23]. Opposite to static linking, which generates a single big application binary, dynamic loading [1] permits complex software to be shipped, delivered, and distributed as a collection of libraries, modules, or components. For low-level languages, such as C/C++ and Rust, these components are implemented as dynamic libraries, also called dynamic-link libraries (`.dll` in Windows) or shared objects (`.so` in Linux). Only when a program starts will its dynamic libraries be integrated to form a runnable application by the dynamic loader. In this way, each dynamic library can be distributed and patched individually without modifying the entire application. As a result, software maintenance cost is greatly reduced while it gains much more flexibility. A study shows that more than 99% of binaries are dynamically linked on Ubuntu [46].

While the dynamic loader's functional structure has been mature and stable for more than one decade, we found it cannot meet the requirements of rapidly developing software and complicated architectures today. Two primary driving factors call out a new infrastructure for extensible and modular transformation on dynamic libraries: (1) the massively increasing number of dynamic libraries used in an application and (2) the emerging diversity of manipulation and operations on dynamic libraries.

Complex commercial software heavily relies on dynamic libraries to decompose a single huge binary into many loosely-coupled, fine-grained modules. This is particularly motivated by two considerations. First, some open source license requires all statically linked code should also be open-sourced. This is so-called "license contamination". GPL license [12] (used by `glibc`) is one such example. Consequently, production software has to use dynamic libraries to avoid "license contamination".

Second, modern software needs frequent updates because of CVE fixes, bug fixes, or adding new features. However, it is painful for vendors to re-compile or link the whole software, and ask customers to reinstall the entire application. Therefore, vendors always break up software into many fine-grained dynamic libraries, and each library can be maintained,

[1] Dynamic loading is also referred to as run-time loading, a mechanism that an application opens, loads, and executes a library by explicitly calling loader interfaces during program execution. As run-time loading shares almost the same backend technology with dynamic loading, throughout this paper, we use dynamic loading to refer to the integrated linking and loading phase when programs are launched.

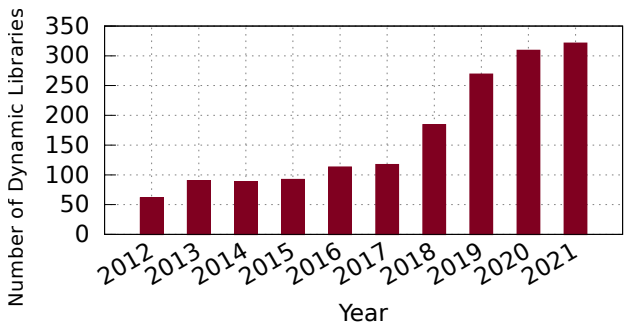


Figure 1: The number of dynamic libraries included in the CUDA Toolkit over the past decade.

updated, or replaced independently. For instance, Figure 1 lists the number of dynamic libraries shipped in the CUDA Toolkit. As it shows, the number has grown rapidly over the last decade. Based on our observation from the industry, this trend will continue in the future.

Along with the growing dynamic library count, the dynamic loader is required to provide more features to make better use of emerging hardware and software technologies. For example, when using recent hardware memory protection (*e.g.* Intel MPK [17] and SGX [16]) to achieve in-process isolation, the dynamic loader has to perform more work. It loads isolated libraries into different memory regions, setups up memory protection and permission properties, and optionally verifies the signature of loaded binaries [7, 14, 37, 38, 40]. Load time code randomization and binary rewriting, provided by the dynamic loader, are widely adopted for profiling, security hardening, and architectural adaptation [51, 53, 54]. Library debloating relies on the dynamic loader to examine and eliminate unused library code from program memory [33, 35]. Control-Flow Integrity (CFI) and Sandbox also require miscellaneous modifications to the dynamic loader, such as analyzing relocation entries and overwriting the entry point [24, 47, 58].

However, the current dynamic loading infrastructure is insufficient and inefficient to offer rich functionalities over a large number of dynamic libraries. This leads to ad-hoc changes to the dynamic loader to satisfy various requirements from different productions. Such customized modifications are incompatible with each other, and cannot be integrated or reused, causing enormous development and maintenance costs. Even worse, the fundamental infrastructure of dynamic loader has been kind of ignored by academia and industry. Thus neither research nor open source community proposes systematic solutions to deal with these issues. For instance, while there are 100+ commits in `glibc` related to the dynamic loader in the last two years, they are almost bug fixes or cleanup without new features developed.

According to our many years' industry experience and realistic production requirements, intrusive and customized modifications cause unacceptable maintenance cost. On the one hand, a large number of source code patches are hard to be accepted by upstream. This also happens to academia work listed above. On the other hand, production departments do not have enough source code level knowledge to maintain patches. Therefore, it is painful for the OS department to maintain many ad-hoc patches and sometimes it has to release different OS distributions with different loaders (along with `glibc`). As a result, it motivates a new infrastructure which satisfies following requirements:

- It offers more functionalities than existing loader.
- It can be flexibly configured for different trade-off and extended to adopt future enhancements.
- Its modifications can be implemented in a modular way that minimizes the effort to align with upstream and fix conflicts due to patch maintenance.

In summary, the issue of current loader design is that it has no interface to allow extensions, thus intrusive modifications cannot be avoided. The loader is historically designed for a few simple functionalities and acts as a "translator". However, now it has to be redesigned, instead of re-engineer, to adopt emerging functionalities and allow future updates in a modular and flexible way, and becomes another platform for application optimization.

We address these challenges by designing iFed, a new infrastructure that achieves extensibility, modularity, and flexibility for dynamic library operations. Our key idea is to organize the iFed as a pipeline of distinct transformation passes instead of a monolithic tool. Each pass only implements some specific manipulation on dynamic libraries to realize its desired functionality, such as security enhancement, memory isolation, or performance optimization. We also design a runnable in-memory format (RiMF) to describe the runtime status and properties of an application and its dynamic libraries (§3.4). RiMF serves as an intermediate representation that every pass operates on, thus different passes are decoupled. By including complete status and information of all dynamic libraries, RiMF further enables iFed passes to do global and aggressive analysis and optimizations. A pass manager orchestrates the series of passes to be applied upon program launch (§3.5). Combined, these features produce the first infrastructure, as far as we know, that satisfies diverse functional requirements without loss of extensibility, flexibility, and modularity.

With various transformation passes plugged in, iFed is able to support much richer features beyond existing dynamic linking and loading. We demonstrate this by implementing two performance optimization passes. The first pass combines the same type of sections from different dynamic libraries into a continuous one, and then leverages hugepages to load the combined section (§3.6). The second one converts relocation branches into direct function calls, thus reducing the overhead of cross-library function calls (§3.7). iFed and its optimization passes are implemented to replace the GNU dynamic loader. We evaluate iFed with a large range of application

benchmarks on different architectures. The results illustrate how iFed optimization passes offer better throughput, latency, and predictability than current dynamic loaders. Without any modifications to the applications in a dynamic website, iFed improves the throughput by 13.3% and reduces the average end-to-end response time by 12.5%.

Our contributions are not only enhancing current loader with some specific optimizations, but also proposing a new infrastructure that is capable to host many other loader features in production. Concretely, the contributions of this paper include:

- We introduce iFed, a pass-based infrastructure for extensible, flexible, and modular transformation on dynamic libraries during load time.
- We design two performance optimization passes in iFed. One pass enables efficient utilization of hugepages by rearrangement and concatenation of multiple libraries. The other pass aggressively eliminates the overhead of cross-library invocations resulting from inefficient relocation.
- We implement iFed infrastructure with the above optimization passes as a drop-in replacement of the default dynamic loader in Linux (`ld.so` in `glibc`). iFed is fully compatible with ld.so and its all interfaces.
- An exhaustive evaluation of iFed on different architectures with a wide range of applications.

The rest of this paper is organized as follows. §2 provides background and motivation for the redesigned dynamic loaders. §3 introduces iFed and discusses its design, while §4 details the implementation of iFed. In §5, we present the performance evaluation of iFed for a wide range of applications. §6 discusses the related work, and §7 concludes.

2 Background and Motivation

2.1 Insufficient Functionality

The basic functionalities of dynamic loading include three parts: (1) library lookup and collection; (2) memory layout preparation; and (3) symbol resolution and name binding. The core jobs to implement these functionalities in existing dynamic loaders are simple. The loader allocates memory and maps libraries into the address space with the given layout specified in library object files. Then it resolves external symbols by populating some lookup tables with the actual memory address. While these steps are just enough to execute programs with dynamic libraries, they are not able to further transform libraries to meet diverse isolation, security, and execution requirements. Thus, many projects have to customize the loader to fulfill their system objectives. We list a few examples here.

- CubicleOS [38] is a library OS that isolates components in MPK protected memory regions, called cubicles. It implements a new cubicle loader who acts as the dynamic loader. The loader is responsible for cubicle creation and component loading. It additionally scans binaries to ensure that there are no any MPK-related operations, and resoles cross-cubicle calls with special trampolines.
- BlankIt [33] is a dynamic loading framework that predicts and loads only the set of library functions that will be used by the application. At load time, BlankIt iterates over all executable's dynamic libraries, wipes out unused functions it predicates, and overwrites these functions with a mispredication trampoline.
- Shuffler [53] patches the loader to support continuous code re-randomization. The modified loader implements constructor prioritization in multiple libraries, and employs binary rewriting to track and update all code pointers.

In summary, while many projects illustrate the necessity and benefit of loader modification, they have to do some redundant work, yet their own work cannot be easily integrated by others. Hence, a new infrastructure for extensible and modular dynamic loading is necessary.

	TLB miss	IPC	99th percentile latency (cycle)	Execution time (s)
`glibc`	1,231,950	1.96	318	6.01
iFed	117,782	2.43	232	4.86

Table 1: Performance comparison between `glibc` and iFed on x86 machine.

2.2 Inefficient Performance

Even worse, current dynamic loaders fail to effectively utilize modern hardware capabilities and global system resources, resulting in sub-optimal performance. A representative case is ineffective hugepage usage.

The current loader loads each dynamic library individually, and within each library, maps code and data section randomly. Thus sections are likely loaded into fragmented memory which only uses small pages (4K) for physical memory. This leads to more TLB miss, slower library function calls, and unpredictable execution time. A better loading strategy is combining the same sections of all libraries into a big one, and loading it into hugepage memory. We study performance penalties incurred by the current loader from `glibc`. On an Intel machine, we conduct a micro-benchmark that simply invokes 100 dynamic libraries, and each library contains only one function accessing memory (full details in §5). Table 1 depicts the micro-architecture impact of (instruction) TLB miss and instruction per cycle (IPC), as well as benchmark results of 99th percentile library function call latency and total execution time. Due to loading libraries with small pages, the benchmark suffers frequent TLB miss, which further leads to slow and unpredictable execution. In contrast, dynamic library concatenation pass in iFed effectively loads libraries

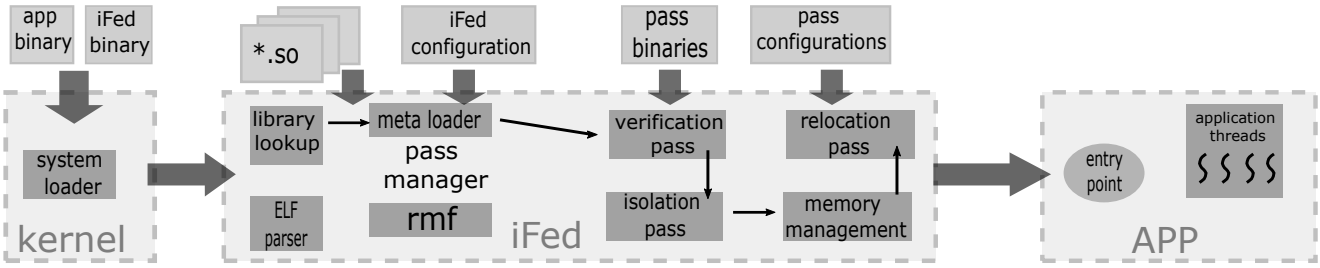


Figure 2: iFed architecture and workflow. The pass manager loads and invokes a series of transformation passes, which interact with RiMF. The workflow of program launch starts from the operating system kernel, which loads both application and iFed binary. After iFed gets the control, it discovers, parses and transforms dynamic libraries and finally boots up the application.

into hugepages, providing an order of magnitude reduction on TLB miss and 23.6% improvement on execution time. Next, we discuss how iFed enables more optimization and transformation of dynamic libraries in a modular and flexible way.

3 iFed Design

3.1 Design Principles

iFed integrates the lessons we learned from the experience of supporting diverse production demands on the dynamic loader. We below outline the key principles, the guidance throughout iFed design.

Extensibility and Modularity (P1). Due to different security or performance considerations, different production always requires a distinct subset of loader features. Therefore, various functionality should be organized in a loosely-coupled way instead of a monolithic implementation. Additionally, iFed should allow applying new features easily without intrusive modification to the loader itself.

Flexibility and Customizability (P2). It is desirable that iFed capabilities can be customized on per-application, customer, or even per-run basis. Such flexibility is important for system managers and end customers to have more control over running applications, opposite to accepting everything from the current loader passively.

Compatibility and Transparency (P3). Compatible with the existing loader interface is critical for iFed to be production-ready. Changes to the loader should be transparent to application developers, and require minimal modification of legacy code. Thus, we aim to design iFed as a drop-in replacement for the existing loader from the beginning.

3.2 iFed Functionality and Usage

As discussed in §2.1, current dynamic loaders cannot keep up with application demands on new functionalities. According to these demands, we summarize the desired features a loader should provide beyond existing ones.

- **Memory management.** The loader should be responsible for memory allocation, library address space layout, and content initialization. This has a large impact on application performance or memory consumption. Some examples of load time memory management are library debloating [33, 35], replaying the profiled hot regions [28], and hugepage optimization (§3.6).
- **Isolation.** The loader is the first place to partition and load different libraries into isolated regions. The customers' strong demand to isolate untrusted or vulnerable third-party libraries paired with the emerging MPK and SGX technologies, motivate the loader to offer more isolation capabilities [7, 14, 40, 50] beyond the traditional read/write/execute permission restrictions.
- **Security enhancement.** The loader is convenient to perform transparent security hardening regardless of running applications. For instance, we can enable CFI or sandbox [5, 24, 47, 58], apply code randomization [26, 51] or perform binary encryption/decryption or signature verification [25, 56].
- **Binary rewriting and execution control.** In addition to traditional relocation, the loader is feasible to perform more advanced binary rewriting and control program execution, such as Shuffler [53] and Egalito [54]. Furthermore, load time transformation is also necessary to migrate applications among heterogeneous environments or offload execution to smart devices [8, 52]. We will discuss a relocation elimination pass in iFed in §3.7.

Current usage of dynamic loader is a mass of interplay among build toolchains, such as compiler and linker. Some configurations and functionalities are scattered in various parts. For example, to prevent GOT overwrite attack [18], the following `gcc` options are widely used: `-Wl,-z,relro,-z,now`. `gcc` passes these options down to the linker, but these options do not take effect until the dynamic loader marks the corresponding memory region as read-only. However, existing usage is not appropriate. We argue that the dynamic loader should be hidden from application developers, but configured and controlled totally by end users or system administrators. This is because customers do not trust that developers properly

build the software to meet their requirements. Thus, iFed consolidates all loader-related operations in one place, and gives the control to end users who actually run the application.

3.3 iFed Architecture

When designing a loader, we should separate functional modules from low-level infrastructure. Functional modules will impact the application run-time behavior and the infrastructure orchestrates these modules. Furthermore, functional modules can be easily replaced or combined without intrusive modifications to the infrastructure and other modules.

We choose pass-based architecture for the loader design. As a result, source code patches are no longer needed, and independent modules with enough semantics can be developed, configured and maintained. The overall iFed architecture is shown in Figure 2. The core component in iFed is a series of transformation and optimization passes that manipulate and transform dynamic libraries for various purposes related to security, isolation, and performance. Each pass is a separate module which can be enabled or disabled independently. Such pass-based modular architecture gives great flexibility and extensibility to users to customize iFed functionality according to their own demands. All passes are managed and controlled by a pass manager (§3.5). Users configure the pass manager to instruct it to construct and execute the pass pipeline. The pass manager also maintains all libraries' in-memory status, and organizes them using RiMF format (§3.4). RiMF is an intermediate representation that is shared by all passes. In this way, passes are able to retrieve global information scatted in many libraries and to perform advanced inter-library transformations. Same as the existing loader, iFed offers other utility components as well, such as library discovery and `elf` parser.

3.4 Runnable In-memory Format

A main goal of iFed is splitting the current monolithic dynamic loader into extensible passes. On the one hand, it is desirable that a pass does not rely on another, thus enabling different passes to be developed and evolve independently. On the other hand, when multiple passes run together, they should be aware of how others transform libraries. Hence, we need a kind of intermediate representation that captures all libraries' status originating from library objects and generated by iFed passes on the fly. Runnable in-memory format (RiMF) is intended to coordinate iFed passes by providing a central place to hold library information at load time.

Passes in iFed do not communicate with each other directly, instead, the shared RiMF is the only interface for library transformation any pass can use. In this way, RiMF hides iFed internal complexity and other pass's implementation details to pass developers. Currently, whenever modifying the dynamic loader to add new features, a developer has to understand most of its codebase, even though much of them are irrelevant. In contrast, all a developer needs to know to write a transformation pass in iFed is the format and properties inside RiMF, and the operations it exposes. iFed maintains a single RiMF image which includes all dynamic libraries a program requires, instead of a separate object file for every library as today. Thus, iFed pass has more opportunities to apply global analysis and optimization. Our dynamic library concatenation pass demonstrates the power of global RiMF. Different from ELF object file which is designed for the dense on-disk format, RiMF rather focuses on load time in-memory representation, such as isolation constraints, memory placement and attributes, and code interposition.

The first-class object in RiMF is the isolation domain, which composes a subset of libraries within the same protection boundary. The actual isolation domain implementation depends on the iFed pass. It could be implemented by MPK, SGX or even device offloading. At the top level, RiMF consists of a list of isolation domains, inter-domain invocations that need to be resolved specially and a global application entry point. Inside each isolation domain, similar to an ELF file, RiMF provides sections, exposed symbols, and relocation records. These information are organized in a set of tables. Primary tables provided by RiMF are: (1) memory-mapping tables which describe library address space layout and memory attributes; (2) symbol tables dealing with symbol definition, binding, reference, and so forth; (3) section metadata tables that associate RiMF sections to original ELF object files. A RiMF section does not contain the actual binary, but maps to one or more ELF sections initially. RiMF varies throughout the iFed transformation pipeline. RiMF exports multiple interfaces to query, insert, modify and commit its internal tables. For example, a pass can update section metadata tables to combine different ELF sections into a new RiMF section. By manipulating symbol tables, a pass is able to remove unused code or override a function call with a customized trampoline. The commit interface is used to apply table modifications to the actual binary, such as interposing them in the library code and loading sections to memory.

3.5 iFed Pass Manager

The iFed pass manager orchestrates transformation passes to operate on RiMF sequentially. The pass manager takes a user-provided configuration file and invokes each pass accordingly. In essence, the pass manager is mainly responsible for two tasks. First, the pass manager maintains the RiMF image and provides interfaces to various passes to query and modify RiMF. Second, the pass manager acts as a meta loader, which loads and executes each transformation pass. Consistent with iFed overall design principle, each transformation pass is also implemented as a dynamic library, which needs to be loaded before execution as well. For simplicity, we reuse the existing `glibc` loader for this minimal meta loader, so any transformation trick is not applied to pass libraries.

Current iFed does not contain a sophisticated scheduling policy for running passes nor supports parallel pass execution. We leave these as future work. Thus, the user has to explicitly deal with pass dependency and pass confliction in the configuration file.

Pass Dependency. In general, passes are not aware of each other because they only use RiMF as the communication medium. However, the order of passes impacts the runtime overhead a lot in some use cases. For example, a binary verification pass is preferred to run as early as possible, so following passes will not waste time on bad libraries. It is also beneficial to place one pass behind another, if it can reuse the analysis result from the previous pass, avoiding repeated work.

Some special cases must be handled carefully. vDSO is one such tricky example. vDSO is a virtual dynamic library (*e.g.* `linux-vdso.so`) inserted into the application by the kernel, but still uses the standard dynamic loading mechanisms. Popular usage of vDSO is mapping some kernel regions into the application's address space, thus some system calls can directly execute on these regions. As a consequence, vDSO libraries must be loaded earlier than any pass that will issue vDSO related system calls. Otherwise, iFed pass itself will fault due to incomplete vDSO even before the application starts running. Similarly, if a pass relies on `malloc` from `libc`, it has to make sure that `malloc` is working properly ahead of the pass execution.

Pass confliction. With more passes integrated together, they are possible to introduce conflict transformations on libraries. Different passes may partition libraries into different isolation domains, or they have opposite optimization objectives. Currently, iFed relies on users to construct the transformation pipeline properly. Automatic dependency extraction and confliction detection will be supported in the future.

Figure 2 demonstrates a potential iFed transformation pass pipeline. All libraries are verified first using security signatures in the first verification pass. Then an isolation pass divides libraries into several isolation domains. Libraries in each domain are loaded into memory, where the memory management pass allocates and sets up memory permissions appropriately. The last binary rewriting pass completes symbol resolution, relocation, and other intent manipulations.

3.6 Dynamic Library Concatenation

Hugepages (superpages) can greatly reduce the address translation overhead, because it eliminates one level page table hierarchy and occupies fewer TLB entries. However, the current loader does not explicitly leverage hugepages. As shown in Figure 3 (a), the current loader individually maps every section in each library into the process's address space. If these sections use a small amount of memory (*i.e.* smaller than the size of a hugepage), the operating system is unlikely to allocate hugepages for them automatically. As a result,

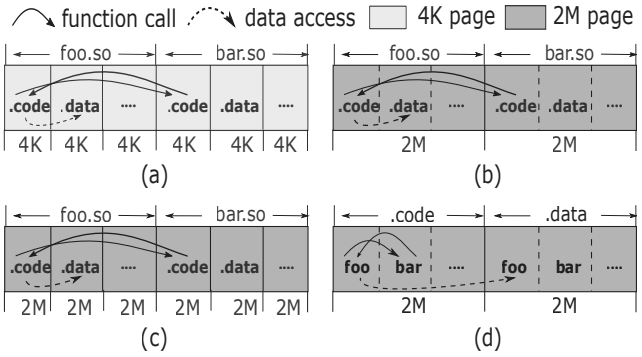


Figure 3: Different hugepage usage schemes for dynamic libraries.

frequent inter-library function calls will trigger more TLB misses, causing expensive page table walk, stalling the CPU instruction pipeline and slowing down applications.

The industry has two approaches to mitigate the impact of high TLB miss, but neither of them is ideal. Figure 3 (b) depicts the first approach, which allocates hugepages to hold all sections in the same library. While this approach reduces the number of used TLB entries, it brings many security vulnerabilities. Since all sections are in the same hugepage, that page should have all read/write/execute permissions required by different sections. For example, `.code` section becomes writeable and `.data` section is executable. Thus, this approach is only used in some closed environments. This, once again, indicates that the loader is capable to alter any policies designated during the development phase, making those policies unreliable. Therefore, the loader should provide capabilities to enforce security policies at load time.

The second method is illustrated in Figure 3 (c), such as the transparent hugepages for file systems proposed in the Linux kernel [27]. In this case, hugepages are used for large sections in each library. While it works well for applications using only a few large libraries, it cannot scale to a larger number of libraries. However, as we discussed in §1, using more and more libraries is the trend for production software, which leads to that such method will be less effective.

In iFed, we design a different approach and implement using in a iFed pass called dynamic library concatenation. The basic idea is intuitive as Figure 3 (d) shows. We collect the same sections, such `.code`, from all dynamic libraries and concatenate them one by one to form a big section. This combined section is large enough to fit in hugepages. More importantly, all the sections share the same memory permissions, so it is safe to place them in the same hugepage. Thanks to RiMF holding all libraries' information, the dynamic library concatenation pass is able to disassemble and rearrange libraries easily.

By combining all libraries `.code` sections into a big one, we might reduce the possible address range used by address space layout randomization (ASLR). To mitigate this security

concern, we have some options. (1) We can concatenate these libraries in random order. [2] (2) Hugepages do not have to be continuous in the virtual address space as long as the original section does not cross two hugepages. (3) We can leverage other code randomization techniques at load time [51] or run time [53], which is easier to employ in iFed.

Another potential negative impact introduced by dynamic library concatenation is library sharing. Dating back to the early days of computing, the motivation for using dynamic libraries is to save limited memory. When multiple running processes require the same library, they only share a single in-memory copy of the libraries. Library concatenation makes the sharing more difficult, as different processes have to use the same set of libraries. However, from our experience, this issue is acceptable for the following reasons. (1) The shared region is mainly the immutable code section. However, the code size of libraries is negligible compared to today's memory capacity. For instance, glibc has around 1.3 million lines of code and its un-stripped binary is only 17 MB, while a common server in the data center has 500 GB memory. (2) Thanks to the customizability of iFed, we can apply library concatenation only to key applications, while other utility or background processes still use the default memory management policy to share dynamic libraries. (3) In some cases, such as edge computing or micro-service, the same process will fork multiple times to serve different customers [36]. Since the forked process has the same address layout, they can share the concatenated library without any problem. (4) In the extreme case where the library must be shared, we align sections from different libraries at the 4K boundary. Thus, the 4K page in the middle of a hugepage can still be mapped to other applications at the cost that others are unable to utilize hugepages.

3.7 Relocation Branch Elimination

An important job accomplished by dynamic loaders is relocation, because the compiler cannot statically resolve cross-library function calls due to lack of address information. After the dynamic loader maps all libraries into process address space, it populates the actual address of unresolved functions in a lookup table. Then every call to a function in a dynamic library first retrieves the address from the lookup table and jumps to that destination. These extra actions result in a trampoline code, which is stored in another table.

Figure 4 (a) shows a simplified execution flow of relocation. The table used to serve address lookup is usually called global offset table (.got) and the procedure linkage table (.plt) saves the trampoline code. When functions in `foo.so` (*e.g.* `foo1` and `foo2`) call the function `bar` in `bar.so`, they call the trampoline (`bar@plt`) instead. The trampoline issues an indirect jump instruction, whose destination address is fetched

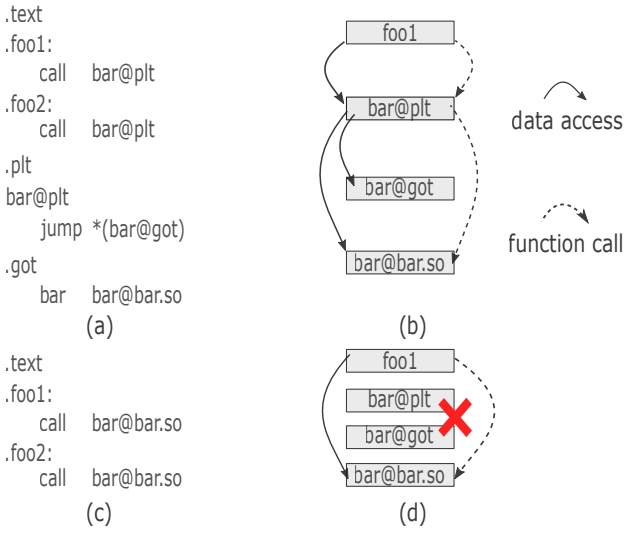


Figure 4: Function call relocation for dynamic libraries. Function `foo1` and `foo2` in `foo.so` call function `bar` in `bar.so`. In (a), function calls are first redirected to `.plt`, and consult `.got` entries to get the destination address, and finally branch to the destination. (b) depicts that the current relocation method incurs three memory access and two code branches. As shown in (c), the relocation branch elimination pass in iFed rewrites the function call sites so that they directly jump to the destination. As a result, only one memory access and code branch is needed in (d).

from an entry in `.got` (`bar@got`). Thus, the execution finally branches to the real address of `bar` (`bar@bar.so`). [3] The above relocation mechanism is applied to every function calls across dynamic libraries, thus incurring pervasive performance overhead. Figure 4 (b) depicts the performance cost in detail.

More executed instructions. Obviously, the single call instruction is expanded to multiple trampoline instructions, consuming more CPU cycles. Even worse, the additional indirect jump puts more challenge on the branch predictor. This is exacerbated by the fact that the trampoline code is not densely packed and `.plt` is often sparsely accessed, leading to more branch misses.

Extra memory access. The existing relocation approach also introduces more memory access. First, `.plt` asks for more memory to store the trampoline. Second, the trampoline needs to load from the extra `.got` memory. More memory access compete for the TLB and cache more frequently. Worse still, they are likely to be evicted from TLB and cache by other data access within the applications, especially in data-intensive scenarios, causing increased function call latency and unpredictability.

[2] Existing loader (*e.g.* `ld.so`) loads libraries in a deterministic way, which is decided by its internal library discovery algorithm according to the dependency information from application binaries.

[3] This simplified execution flow omits some complexities. .got entries are initially populated with a pointer to a loader's own resolver function. So when a library function is invoked at its first time, it branches to the resolver function, which then updates the `.got` entry using the actual address. This also requires additional instructions to be patched into the trampoline.

However, there are no practical solutions to eliminate these performance penalties. Switching to statically linked libraries is not always feasible as discussed in §1, and some hardware methods [1] are not available in production due to architectural modifications. It is also difficult to replace the relocation mechanism in the current dynamic loader with little effort.

Thanks to iFed, we have a chance to insert an optimization pass to reduce the relocation cost in an extensible manner. We design the relocation branch elimination pass for this purpose. The key idea inside relocation branch elimination is pretty intuitive. As shown in Figure 4 (c), we can directly rewrite the `call` instructions to replace their target address using the address of library functions, instead of the address of the trampoline in `.plt`. The performance gain is obvious. We eliminate the extra two memory access and one instruction branch as shown in Figure 4 (d). As a result, we essentially achieve the performance of static linking on top of dynamic libraries. Despite its simple idea, we have to deal with instruction decoding, relocation sites management, and other implementation issues carefully. Implementation details are discussed in §4.

Rewriting instructions causes it more difficult to share libraries among applications, since they have to be organized in the exact same address space layout. Thus, the relocation branch elimination pass is preferred to be used in environments with sufficient memory. Another challenge that needs to be overcome is the distance restriction of a relative branch. When using relative addressing mode, the CPU has restrictions on the distance between the call site and the target address. [4] Therefore, only rewriting the target address is not always possible if the library functions are loaded far from the call sites. This issue can be handled in multiple ways. (1) When combined with the dynamic library concatenation pass, it is rare that the distance exceeds the architectural constraint. (2) We can change the relative addressing to absolute addressing mode at the cost of an extra instruction to load the address into a register. This change can be done by recompiling the code or rewriting the instructions by the loader. For instance, the Linux kernel module loader rewrites the instructions when detecting the constraint violation. (3) For the call sites that are far away from the target function, we can fall back to the existing relocation method using `.plt` and `.got`.

3.8 Discussion and Summary

The pass-based architecture enables iFed to accommodate much more load time technologies and functionalities. However, we do not argue that our architecture is the only or best way to design a loader. Other methods are possible, such as "Linux kernel module" or "systemd service unit" approach. This is an open and new research area, and researchers are welcome to investigate more. iFed also brings side effects to program launch time and binary size, and we discuss these trade-offs below.

Loading Time. While iFed infrastructure itself does not introduce additional overhead to program launch, boot time will increase as more iFed transformation passes are enabled. End users have to make the judgment on the trade-off between longer loading time and securer or faster application in run time. According to our experience so far, the increased loading time in iFed is acceptable. This is because (1) For applications that already require a modified loader to provide new functionalities, they do not suffer more extra launch costs after switching to iFed; (2) For long-running services, such as web server and database, the one-time overhead during the startup is always negligible; and (3) For short-lived tasks in high churn environments, we can explore process template and in-memory caching technology [36] to fork processes from an initialized template, thus all forked processes will bypass iFed loading phase and its associated overhead. We study how our dynamic library concatenation and relocation branch elimination passes impact loading time in §5.

Binary Size. As some iFed transformation passes may need extra binary information to perform in-depth analysis, it is likely to bloat the application binaries. For example, the relocation branch elimination optimization requires the linker to retain all relocations in the executable file, resulting in larger binaries. While it is possible to scan the binary to re-generate these information, it is not wise to waste time on these redundant work. So far, the bloated binaries are not a big deal given the current massive persistent storage, but we argue the ELF-based binary scheme can be improved in the following senses. First, developers should keep relevant binary information (*e.g.* data generated by static analysis or bitcode of LLVM IR) as much as possible to reflect more comprehensive semantics close to the source code, instead of throwing them away at build-time and hiding them from the users. It is the user who makes the decision whether these information should be stripped at deploy- or install-time. Second, while iFed uses ELF-based objects for compatibility now, it is better to have a different object file format in iFed to match the pass-based structure and RiMF image. Particularly, object files can be disassembled into per-pass pieces, and these pieces can be fetched, trimmed, or analyzed through per-pass configuration. These improvements are left as future work.

Summary. We summarize how iFed resolves the issues discussed in §1 based on our design principles. Organizing iFed with a collection of transformation passes inherently achieves modularity (P1). Passes do not directly interact with each other, but rely on the pass manager to mediate and operate on RiMF image as the only interface for collaboration. New passes are easily plugged into iFed, which significantly improves extensibility in iFed (P1). Therefore, vendors do not need to randomly modify the loader nor maintain multiple

[4] This is because only a subset of bits in the branch instruction is available to encode the address. For example, x86 limits the range as $\pm$2 GB, while ARM has a limitation of $\pm$128 MB.

versions to satisfy customers' different demands, and in the meantime, customers are able to enjoy more features for free. Since users can choose which pass to be used in iFed via iFed configuration, they can flexibly construct transformation pipelines to customize the application at load time (P2). Paired with the iFed's capability to transform libraries with a global view, customers are flexible to determine the trade-off among security, isolation, and performance. iFed is implemented to be compatible with the current loader, so no application modification is required (P3). More importantly, iFed enables another level of transparency for system administrators. For example, managers can insert a default security enhancement pass to iFed, regardless of if applications are built with security options.

4 iFed Implementation

Figure 2 depicts the typical workflow of iFed during program launch. A program's binary is first loaded by the operating system, which then loads the dynamic loader's binary if necessary. Next, the kernel returns to user space and hands over the control to iFed. After discovering all required libraries, iFed invokes the pass manager with an initial RiMF image which simply contains all libraries in a single isolation domain. Based on the iFed configuration, the pass manager loads and executes each pass in sequence. Finally, iFed invokes the application's entry point and completes the loading phase.

Compatibility. Current iFed is implemented on top of `glibc` 2.28. We reuse some utility components, such as library discovery and ELF parser from the `glibc`. As a result, iFed is able to load unmodified ELF binaries and supports common loader extensions, such as `LD_PRELOAD`. For compatibility, the existing dynamic loader (*i.e.* `ld.so`) is organized as a special fake pass in iFed. Linux allows an application to specify the dynamic loader it will use. Thus, we use this facility to enable the usage of iFed within applications.

Dynamic Library Concatenation. In this pass, we collect the same sections from all libraries and pack them into continuous memory backed by hugepages. To save memory, the last page is converted to small pages if less than 64 KB memory is occupied. While the implementation is intuitive, we must fixup the global variable access. Global variables are always accessed via offset, which is the difference between the address of the accessing instruction and the variable itself. For example, in Figure 3 (a) and (d), the offset between the `.code` and `.data` section within the `foo.so` is changed due to the rearrangement. Thus accessing variables in the `.data` section is broken. Our current solution is to instruct the compiler to emit all the symbol access information (*e.g.* using `-emit-relocs` options in `gcc`), and to fix the offset during the pass execution. The book-keeping infomation inside iFed is also updated according to the finalized address, so as to serve run-time loader interfaces, such as `dlsym()` and `dladdr()`. We only rearrange the libraries which are position independent.

Relocation Branch Elimination. This pass rewrites the branch instructions so that they do not need indirect jump based on `.plt` and `got`. First, we identify all branch instructions from the relocation records. Each record saves the position of the instruction and the remote symbol it references. The symbol could be either a function or a variable. Then, we find the actual address of the symbol and modify the instruction to use the address instead. Modifying instructions is architecture-dependent. We further optimize the function pointer invocations. In case of the function address can be determined at the loading time, we substitute the function pointer with the actual function.

5 Evaluation

Our evaluation goals include:

- Illustrate the effectiveness of dynamic library concatenation and relocation branch elimination pass using micro-architecture statistics.
- Understand the applicability of iFed along with our optimization with a wide range of applications.
- Assess the generality when running iFed on different hardware architectures.

Setup. We evaluate iFed on two architectures. The first one is two 26-core sockets Intel(R) Xeon(R) CPU @ 2.3GHz, with hyper-threading enabled, resulting in 104 cores in total. The other is ARM Kunpeng-920 CPU @ 2.6GHz with four NUMA nodes, and each node has 24 cores. All experiments run on openEuler 20.03 [30] based on Linux 4.19 kernel. We compare iFed with the system default dynamic loader, `ld.so` in `glibc 2.28`.

5.1 Micro-benchmarks

We conduct a set of micro-benchmarks to evaluate the performance improvement of library concatenation and relocation branch elimination passes in iFed. Each test calls functions provided by a configurable number of dynamic libraries, and each function accesses a certain amount of memory. Figure 5 and Figure 6 study the impact of different library counts and working set sizes in the library function, respectively. All tests are run 500K iterations on the Intel machine. We compare four different implementations, (1) `glibc` – the system default dynamic loader. (2) iFed-`hugepage` – iFed with only dynamic library concatenation pass. (3) iFed-`relocation` – iFed with only relocation branch elimination pass. (4) iFed– iFed with both optimization passes.

Micro-architecture Impact. Figure 5 (a) shows the number of misses in instruction TLB. With more libraries involved, the total iTLB miss grows rapidly. `glibc` incurs the most iTLB miss because it uses 4K pages to load libraries and runs out of the limited number of iTLB entries. iFed-

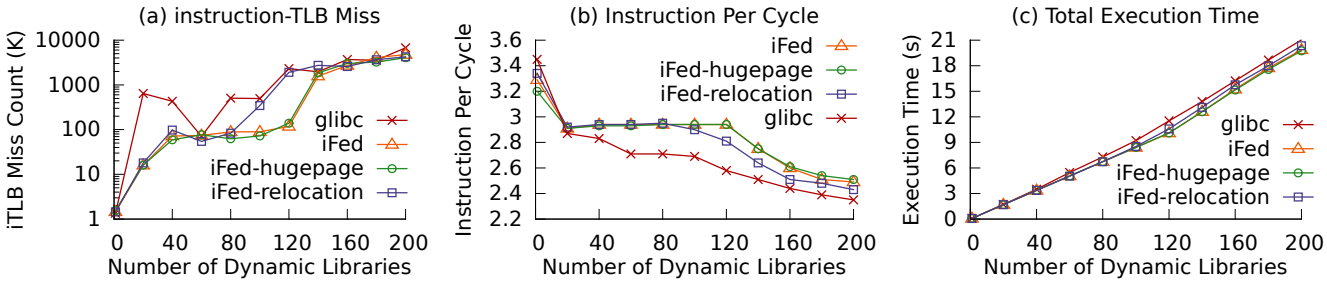


Figure 5: Micro-benchmarks: the working set is fixed at 256 KB.

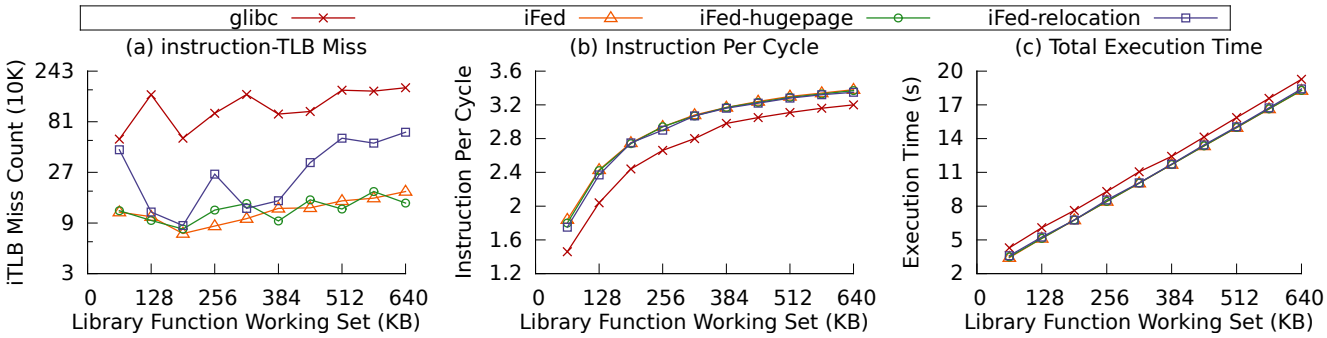


Figure 6: Micro-benchmarks: the number of dynamic library is 100.

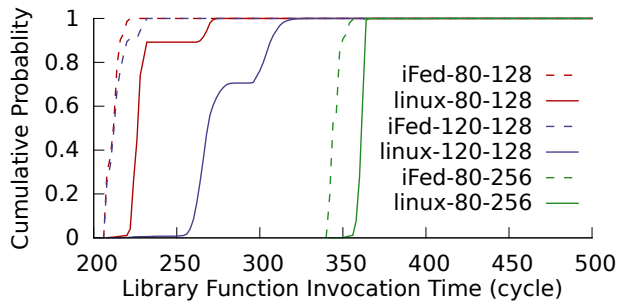


Figure 7: Library function invocation latency distribution.

`relocation` has little difference with `glibc` as it uses 4K pages, too. However, iFed-`hugepage` and iFed perform much better than `glibc` (note the log scale of y axis). Thanks to the usage of hugepage, they reduce the iTLB miss by an order of magnitude when the library count is smaller than 160, and by 40% with larger library counts. Less TLB miss leads to higher IPC as shown in Figure 5 (b). In addition to TLB miss, fewer code branches also decrease IPC. Thus, `glibc` has lower IPC than iFed-`relocation`, and iFed performs the best after integrating both optimizations. While the purposed optimizations almost work on `.code` sections, they also get benefits with a varied amount of data access as shown in Figure 6. With more data access, they compete for the cache and TLB when shared with `.code`, `.plt` and `.got` sections. This

`glibc`	iFed-`hugepage`	iFed-`relocation`	iFed
1.42 ms	5.96 ms	7.02 ms	9.07 ms

Table 2: Loading time overhead comparison. These are the cost to load a redis server which has 36195 relocation sites.

is illustrated in Figure 6 (a) where iFed-`relocation` triggers less TLB miss than `glibc`. In Figure 6 (b), IPC increases with the larger working set, as the memory access dominates the program execution. However, `glibc` performs worse than all iFed variants.

Latency Analysis. The improvements on micro-architecture further lead to the reduction in total execution time as depicted in Figure 5 (c) and Figure 6 (c). All execution time rise linearly with the test scale, but iFed runs faster than `glibc` in all cases. For instance, with 200 libraries and a 256 KB working set, iFed is 6% faster than `glibc`. More importantly, due to less TLB miss and branch, the predictability of library function invocation is improved a lot. To better understand the latency of library function calls, Figure 7 presents a CDF of function call latencies under different configurations. From the results, we observe that `glibc` has higher tail latency than iFed. For the 99th percentile latencies under the three configurations, iFed has improvements of 19%, 27%, and 3%, respectively.

Loading Time Discussion. Since iFed incorporates more functionalities, it inevitability slows down the time to launch a

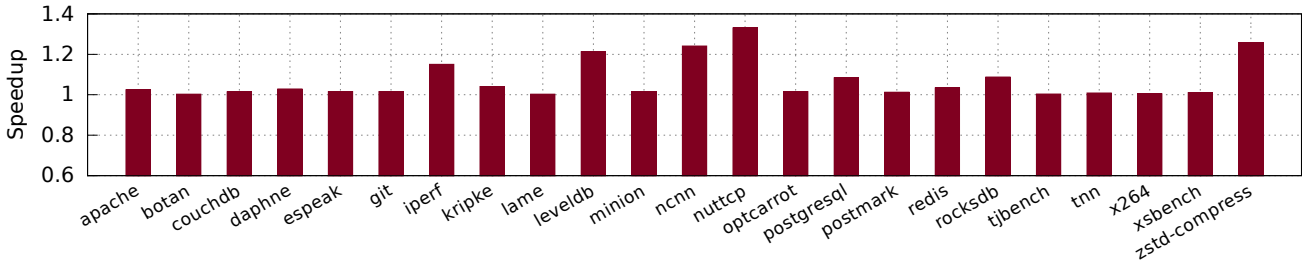


Figure 8: Phoronix test suite on ARM physical machine.

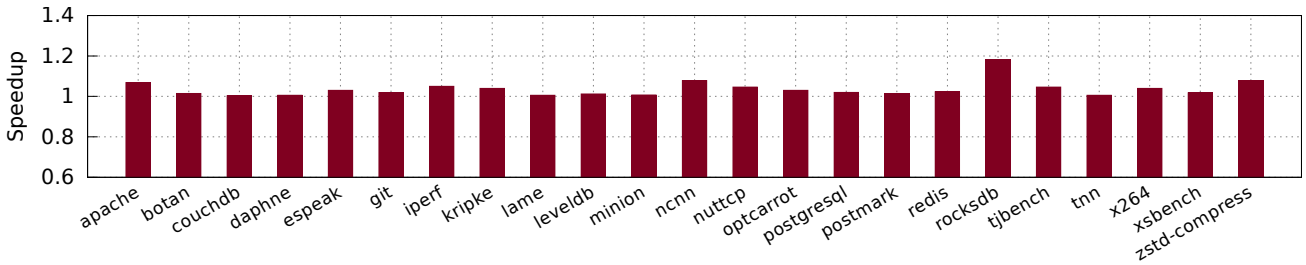


Figure 9: Phoronix test suite on x86 virtual machine.

program. Table 2 reports the loading time spent in the interval from the `exec` system call to the program's `main` function, when loading a redis server. As discussed in §4, the current iFed contains the `glibc` loader for compatibility, thus the result differences indicate the overhead of iFed optimization passes. Dynamic library concatenation overhead mainly comes from memory movement. The cost of relocation branch elimination depends on the number of relocation sites that has to be rewritten, thus it may incur a larger overhead.

5.2 Application Benchmarks

We evaluate iFed with Phoronix test suite v10.4.0 [42], which has a wide range of common applications and realistic workloads. We selected 23 applications from multiple domains that stress different components in the system, such as memory (`zstd-compress`), processor (`botan`), disk (`postmark`) and network (`iperf`). These tests also cover different running forms of multi-process, single- and multi-thread. We run these tests in two environments, the ARM Kunpeng server and a 60-core KVM-based virtual machine hosted in the Intel machine, because VMs are popularly deployed to hold applications today. Hardware virtualization is enabled, and the VM is configured with 128G memory. The guest OS is also openEuler 20.03 based on Linux 4.19 kernel. We enable all optimization passes in iFed, and report the average performance speedup compared to `glibc` in Figure 8 and Figure 9. The data is gathered from the built-in performance comparison tool in Phoronix. Since better hugepage usage and eliminated `.got/.plt` indirection in iFed will improve many tightly correlated micro architecture factors, we use `perf` to measure some typical CPU events for each benchmark. Table 3 lists the percentage of TLB miss reduction, branch miss reduction, and IPC improvement compared to `glibc` on both ARM and Intel testbeds.

Whether an application can get benefits from iFed depends on its bottleneck. For computing intensive applications who do not suffer from TLB miss or branch mispredictions, iFed keeps the same performance with `glibc`. For example, `botan` is a C++ crypto library and the benchmark measures the performance of many cryptographic algorithms. iFed has less than 1.5% performance difference with `glibc` in all test cases. As shown in Table 3, iFed has a negligible impact on IPC. `xsbench` tests a key computational kernel of the Monte Carlo neutronics application OpenMC. iFed does not reduce branch misses on ARM, thus the performance difference between iFed and `glibc` is less than 2%.

When the application is memory bound and its data compete for the shared TLB and cache with the code, iFed is able to mitigate the interference and improve the performance. For instance, the `zstd-compress` benchmark compresses and decompresses a 1 GB Linux kernel image. iFed reduces the number of TLB misses by 16.56% and 19.4% on Intel and ARM machine, respectively. Please note that our dynamic library concatenation deals with both `.code` and `.data` section, thus iFed does not only reduce iTLB misses. As a result, iFed speedups the benchmark by 7.3% on Intel and 25.7% on ARM.

For complicated applications that have complex function call patterns across libraries or use many dynamic libraries, iFed can boost their performance. For example, `leveldb` from Google gets 1.1% and 21.2% better performance on Intel and ARM platform, respectively. On ARM, `glibc` in-

curs 10^9 instruction TLB misses, while iFed just incurs 10^5 iTLB misses! `ncnn` is a mobile neural network inference framework developed by Tencent. Its IPC is improved by 3.04% and 6.36% on Intel and ARM platform respectively, and correspondingly iFed gets 7.7% and 24% overall better performance. `iperf` and `nuttcp` have a large performance boost because both benchmark server and client are loaded by iFed.

In some cases, iFed shows large relative improvements on perf events while has little impacts on the benchmark performance. That is because the event's absolute numbers are so small that slight variations of the event result in large percentage difference. For example, on the Intel machine, `optcarrot` shows 10.45% less TLB misses while its performance is only 2.9% better. After examining the absolute number of TLB misses, we found there are only 1.8 million misses with `glibc` and iFed lowers it to 1.6 million. Those numbers are several orders of magnitude smaller than those in other memory intensive benchmarks. Another example is branch miss reduction in `botan` benchmark on the ARM platform. `botan` experiences around 323K and 311K branch misses under `glibc` and iFed respectively. Despite 3.65% reduction in branch misses, iFed does not have speedup over `glibc`.

To further validate our results, we analyse the `rocksdb` benchmark on the Intel VM in depth. The benchmark contains 3857 `.got` entries and 8153 `.plt` entries, and 94327 relocation sites point to these entries. With `glibc`, 15.6% of total cycles are spent on page table walk due to TLB misses, while this ratio is reduced to 10% after iFed optimization. Relocation branch elimination pass contributes 6% improvement, and dynamic library concatenation pass continues to improve 10%, leading to an overall improvement of 18%. We also tested a statically linked version which performs 9% better than the dynamic one with `glibc`. This improvement is less than iFed with the hugepage optimization, but is better than iFed with relocation elimination since static linking has more chance to apply link-time optimization.

In general, we do not observe the loading time overhead causing performance degradation even for the benchmarks which need to frequently boot up and shut down the test programs. On the Intel virtual machine, compared to `glibc`, the average TLB miss is reduced by 8.58%, the average branch miss is reduced by 3.28%, and the average IPC is improved by 3.02%. iFed is 3.7% better than `glibc` on average and achieves 18% maximum improvement. On the ARM physical

benchmark name	x86			ARM		
	tlb miss	branch miss	instruction per cycle	tlb miss	branch miss	instruction per cycle
apache	12.27%	8%	4.98%	5.44%	1.82%	0%
botan	8.01%	3.13%	0.09%	0.05%	3.65%	0%
couchdb	3.86%	0.79%	4.64%	4.94%	0%	0%
daphne	8.25%	5.56%	2.22%	2.15%	4.25%	4.47%
espeak	12.6%	1.33%	0.26%	32.04%	0.07%	0.37%
git	3.85%	6.71%	2.54%	1.36%	0.46%	3.09%
iperf	7.58%	5.03%	5.73%	27.95%	3.91%	19.44%
kripke	4.56%	10%	4.31%	17.79%	1.12%	5.41%
lame	7.4%	11.52%	1.17%	18.1%	0.7%	1.19%
leveldb	3.15%	1.37%	4%	34%	5.29%	32.43%
minion	13.66%	0.71%	1.63%	1.01%	0.37%	1.54%
ncnn	7.98%	5.03%	3.04%	37.05%	2.87%	6.36%
nuttcp	3.12%	3.03%	5.92%	34.55%	6.95%	56.52%
optcarrot	10.45%	1.39%	3.85%	0.24%	1.65%	1.49%
postgresql	5.93%	2.21%	1.56%	10.33%	2.4%	4.05%
postmark	3.3%	0.7%	-0.47%	9.97%	0.63%	0.85%
redis	6.4%	1.01%	1.54%	12.78%	2.12%	2.23%
rocksdb	35.9%	4%	13%	13.52%	2.71%	8.16%
tjbench	14.82%	-0.34%	3.59%	2.83%	0.1%	0.61%
tnn	1.4%	1.09%	0.57%	2.69%	1.11%	1.26%
x264	1.99%	0.64%	1.28%	1.88%	0.51%	0.62%
xsbench	4.21%	1.27%	1.78%	9.96%	-0.58%	0%
zstd-compress	16.56%	1.32%	2.25%	19.4%	0.48%	18.52%
average	8.58%	3.28%	3.02%	13.04%	1.85%	7.33%

Table 3: Application benchmarks: percentage of TLB miss reduction, branch miss reduction, and IPC improvement..

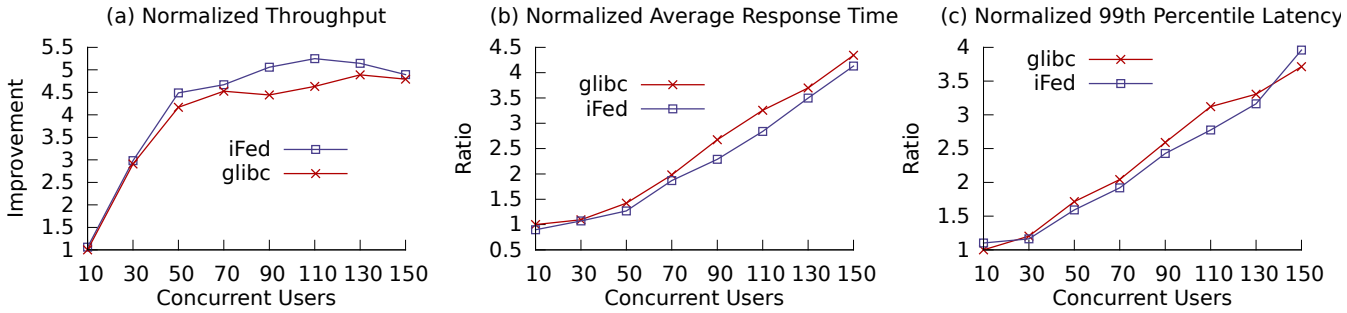


Figure 10: Dynamic web serving performance. All data are normalized to the result of 10 concurrent users with `glibc`. (a) shows the throughput across all operations, the higher the better; (b) shows the average response time of `postwire` operation (similar to posting a tweet), the lower the better; (c) shows the 99th percentile latency of `postwire` operation, the lower the better.

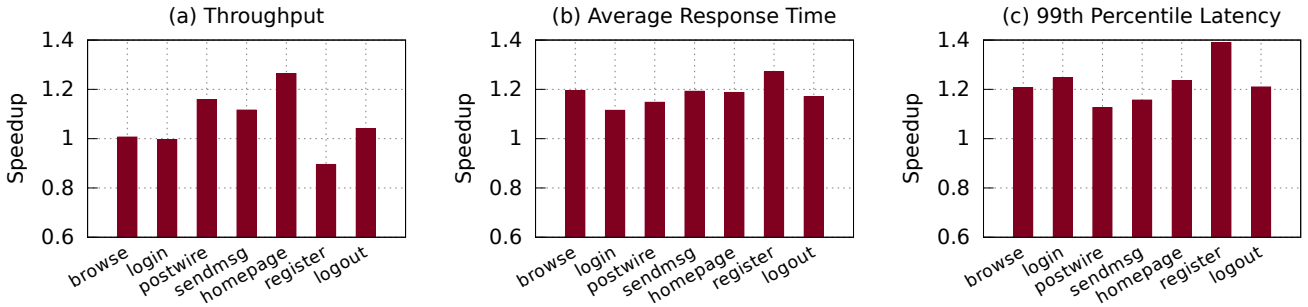


Figure 11: Performance of each operation with concurrent 110 users. All data are the speedup ratio normalized to the `glibc`, the higher the better.

machine, iFed reduces 13.04% TLB miss, lowers the branch miss by 1.85%, and improves the IPC by 7.33%, on average. The average speedup is 7% and the largest improvement is 33%. In most cases, iFed achieves a larger improvement on the physical machine than the virtual machine. This is because VMs have an extra address translation layer. Thus if the host OS allocates small pages to the guest OS, iFed will get less benefit by enabling hugepage in the guest OS.

5.3 Web Serving

Finally, we evaluate iFed in a system-wide scenario with a web serving benchmark from Clousuite [10]. This benchmark is a dynamic website hosting a production-quality social networking engine. Since the current Clousuite is not supported on ARM architecture, we port it to our ARM machine, and upgrade its components to newer versions. Particularly, we use nginx 1.16.1, mysql 8.0.17, PHP 7.2.10, and elgg 3.0.7. We run the client and server on two ARM machines under the same ToR switch. The client simulates multiple users who browse the website and issue different operations, such as register, login, and send messages to a friend. These operations are mixed in a distribution that favors common operations (*e.g.* send a message and post a tweet), while containing fewer login/logout operations. Each test case runs 5 minutes, and Clousuite collects the throughput and response time.

Figure 10 shows the normalized performance with various simulated concurrent users. The efficiency is seen in the improved throughput, reduced response time, and tail latency. The performance keeps increasing with more users until the system is saturated. For the peak performance, iFed has 13.3% higher throughput, 14.7% smaller average response, time and 12.5% lower 99th percentile latency. Figure 11 shows the detailed performance statistics of each operation with 110 concurrent users. iFed is better than `glibc` in most cases. For the throughput of `register` operation, iFed is lower because the client issues less `register` operation due to the probabilistic workload distribution. This is also confirmed by the reduced response time from Figure 11 (b) and (c). To summarize, these results demonstrate that optimizations in iFed are effective in the realistic multi-application environment.

6 Related Work

Loader modification and improvement. Many projects have to modify the loader to achieve their specific goals, even though the loader is not of their research contribu-

tion. However, the current dynamic loading infrastructure is neither flexible nor extensible to accommodate those modifications, causing their research to have limited applications in the industry. When utilizing MPK [14, 38, 50] or SGX [7,13,32,37,40,44,60] for stronger isolation or security, most works have to modify the loader to be aware of such isolation facility. Besides hardware-assistant isolation, software implementation also require to coordinate with loader, such as sandbox [5,6,24,55] and CFI [22,25,47,58]. It is necessary to change the loader to support program migration and execution on remote, heterogeneous, or smart devices [3, 8, 52]. Kard [2] leverages MPK for per-thread memory protection to implement a dynamic data race detector, which uses a custom loader to handle global variables. Shuffler [53] continuously re-randomizes code locations in a separate thread, but requires a small loader patch for bootstrap. With iFed, these modifications will be made easily and further reused across different projects.

Agrawal et al. propose a speculative hardware mechanism to avoid executing relocation trampolines [1], while we provide a pure software approach to eliminate relocation overhead in iFed. Stephen Kell et al. describe the formal semantics for static linking [19]. As iFed decouples a monolithic dynamic loader into smaller pieces, we expect a similar formal method can be applied to dynamic linking as well.

Load time technologies. There is a large body of research focusing on load time technology. Paschalis Mpeis et al. introduce a capture and replay mechanism [28] that detect and profile hot code regions, and optimize them offline. Instead of the original code from binary, these captured and optimized hot regions are fed into the loader to replay. Egalito [54] is a binary transformation framework that supports dynamic analyses or code-generation at load time. Load time binary stirring [51] randomly reorders some code sections and repairs code pointers accordingly. ASLR-Guard [26] contains a dynamic loader, which decouples code sections from data sections and encrypts some sensitive regions. Library debloating [33,35] is a type of load time optimization that loads only the set of library functions that will be used at each library call site within the application at runtime. iFed provides a platform to explore and integrate broader load time technologies. Wei Dong et al. propose a holistic dynamic linking and loading mechanism in networked embedded systems to generate minimal code size [9].

Loader on new system and architecture. Since dynamic loader is a basic toolkit, it has to be rewritten whenever a new system or hardware comes. For example, RedLeaf [29] is a rust-base OS with a new abstraction, called Domains, for lightweight isolation, and supports dynamically loaded Domains. CARAT [41] allows programs to run efficiently in a physical address space and needs a loader to collaborate properly. Different loaders are also implemented within different system architectures, such as microkernel [20,49], unikernel [43] or LibOS [4,34,45,59]. Similarly, the loader is always needed to be updated to explore new hardware features for isolation [39], security [31], communication [48], container [57], and embedded device [21]. With the help of iFed's modular design, we are able to extract the system agnostic or architecture independent parts and reduce the porting effort.

7 Conclusions

We introduce iFed, an infrastructure for dynamic library transformation. While iFed is compatible with the current dynamic loader, its function goes beyond the traditional dynamic linking and loading. By a pass-based architecture and RiMF, iFed can provide much richer functionalities over isolation, security, and optimizations in a flexible, extensible, and modular way. We demonstrate the extensibility of iFed by implementing two optimization passes. One pass reduces TLB miss and improves IPC because of the effective usage of hugepages. The other pass rewrites the call sites to eliminate function relocation overhead. Modularity and extensibility are crucial to reducing the development, deployment, and maintenance costs of today's complicated system software. We believe it is an open research area to investigate modular design in many other monolithic system software, not just in the loader.

Our evaluation shows optimizations in iFed improve performance and predictability for a wide range of applications on multiple architectures and platforms. On an ARM physical machine, iFed achieves up to 33% speedup, and on an Intel virtual machine, iFed gets a maximum improvement of 18%. In a complex dynamic website that requires collaboration among multiple applications, iFed improves the throughput by 13.3% and achieves a 12.5% reduction of end-to-end 99th percentile latency. More importantly, iFed boosts the performance transparently with no application changes. Building on both customers' demands from industry and load time technology advances from academia, the dynamic library manipulation infrastructure is a promising area of research. We believe that iFed paves the first example of a new generation of dynamic loaders for integrating research advancement of load-time transformations and technologies.

Acknowledgments

We sincerely thank our shepherd Andreas Haeberlen for his insightful feedback. We are grateful to the OSDI anonymous reviewers for their valuable comments and suggestions. We thank the EulerOS team at Huawei for their contributions to this work, including but not limited to, Zixian Liu, Bin Wang, Sirui Liu, Pan Zhang, Lin Fu, Xiangyang Yu, Yanchao Yang, Chao Liu, Danni Xia, Jiaqi Yang, Yining Shen, Tianxiong Lu, Haomin Cai, Wei Du, and Guiping Zhang.

References

[1] Varun Agrawal, Abhiroop Dabral, Tapti Palit, Yongming Shen, and Michael Ferdman. Architectural support for dynamic linking. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, 2015.

[2] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. Kard: Lightweight data race detection with per-thread memory protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[3] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, 2017.

[4] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, 2013.

[5] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementatio (OSDI'12)*, 2012.

[6] Mihai Bucicoiu, Lucas Davi, Razvan Deaconescu, and Ahmad-Reza Sadeghi. Xios: Extended application sandboxing on ios. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (Asia CCS'15)*, 2015.

[7] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (ATC'17)*, 2017.

[8] Shenghsun Cho, Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. Flick: Fast and lightweight isa-crossing call for heterogeneous-isa environments. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*, 2020.

[9] Wei Dong, Chun Chen, Xue Liu, Jiajun Bu, and Yunhao Liu. Dynamic linking and loading in networked embedded systems. In *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, pages 554–562, 2009.

[10] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, 2012.

[11] Michael Franz. Dynamic linking of software components. *Computer*, 30(3):74–81, March 1997.

[12] Free Software Foundation (FSF). Gnu lesser general public license, `https://www.gnu.org/licenses/lgpl-3.0.html`.

[13] Lukas Giner, Andreas Kogler, Claudio Canella, Michael Schwarz, and Daniel Gruss. Repurposing segmentation as a practical lvi-null mitigation in sgx. In *31st USENIX Security Symposium (USENIX Security'22))*, 2022.

[14] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (ATC'19)*, 2019.

[15] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software-Pratice and Experience*, 21(4):375–390, April 1991.

[16] Intel. Intel(R) Software Guard Extensions, `https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html`.

[17] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual.*

[18] Seunghoon Jeong, Jaejoon Hwang, Hyukjin Kwon, and Dongkyoo Shin. A cfi countermeasure against got overwrite attacks. *IEEE Access*, 8:36267–36280, 2020.

[19] Stephen Kell, Dominic P. Mulligan, and Peter Sewell. The missing link: Explaining elf static linking, semantically. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, 2016.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood.

Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, 2009.

[21] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, 2014.

[22] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[23] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[24] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (ATC'14)*, 2014.

[25] Yan Lin, Xiaoyang Cheng, and Debin Gao. Control-flow carrying code. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications (AsiaCCS'19,)*, 2019.

[26] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[27] LWN.net. Transparent huge pages for filesystems, `https://lwn.net/Articles/789159/`.

[28] Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather. Developer and user-transparent compiler optimization for interactive applications. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*, 2021.

[29] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[30] openEuler. `https://openeuler.org`.

[31] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, 2020.

[32] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, 2018.

[33] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: Getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*, 2020.

[34] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, 2011.

[35] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security'18))*, 2018.

[36] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-grained isolation for scalable, dynamic, multi-tenant edge clouds. In *2020 USENIX Annual Technical Conference (ATC'20)*, 2020.

[37] Vasily A. Sartakov, Daniel O'Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. Spons & shields: Practical isolation for trusted execution. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'21)*, 2021.

[38] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[39] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[40] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single

enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[41] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. Carat: A case for virtual memory through compiler- and runtime-based address translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*, page 329–345, 2020.

[42] Phoronix Test Suite. `https://www.phoronix-test-suite.com/`.

[43] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'20)*, 2020.

[44] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter R. Pietzuch. rkt-io: a direct I/O stack for shielded execution. In *Sixteenth European Conference on Computer Systems (EuroSys'21)*, 2021.

[45] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, 2014.

[46] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern linux api usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*, 2016.

[47] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[48] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. Direct inter-process communication (dipc): Repurposing the codoms architecture to accelerate ipc. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*, 2017.

[49] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. SPeCK: a kernel for scalable predictability. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15)*, 2015.

[50] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. Secure and efficient in-process monitor (and library) protection with intel mpk. In *Proceedings of the 13th European Workshop on Systems Security (EuroSec'20)*, 2020.

[51] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, 2012.

[52] Yaron Weinsberg, Danny Dolev, Tal Anker, Muli Ben-Yehuda, and Pete Wyckoff. Tapping into the fountain of cpus: on operating system support for programmable devices. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'08*, 2008.

[53] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[54] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[55] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy (S&P'09)*, 2009.

[56] Hansen Zhang, Soumyadeep Ghosh, Jordan Fix, Sotiris Apostolakis, Stephen R. Beard, Nayana P. Nagendra, Taewook Oh, and David I. August. Architectural support for containment-based security. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[57] Hansen Zhang, Soumyadeep Ghosh, Jordan Fix, Sotiris Apostolakis, Stephen R. Beard, Nayana P. Nagendra, Taewook Oh, and David I. August. Architectural support for containment-based security. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[58] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security'13)*, 2013.

[59] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. Kylinx: A dynamic library operating system for simplified and efficient cloud virtualization. In *USENIX Annual Technical Conference (ATC'18)*, 2018.

[60] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. Mptee: Bringing flexible and efficient memory protection to intel sgx. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, 2020.

Application-Informed Kernel Synchronization Primitives

Sujin Park[†] *Diyu Zhou* *Yuchen Qian* *Irina Calciu*[*] *Taesoo Kim*[†] *Sanidhya Kashyap*

EPFL [†]Georgia Tech [*]Graft

Abstract

Kernel synchronization primitives are the backbone of any OS design. Kernel locks, for instance, are crucial for both application performance and correctness. However, unlike application locks, kernel locks are far from the reach of application developers, who have minimal interpolation of the kernel's behavior and cannot control or influence the policies that govern kernel synchronization behavior. This disconnect between the kernel and applications can lead to pathological scenarios in which optimizing the kernel synchronization primitives under one context, such as high contention, leads to adversarial effects under a context with no lock contention. In addition, rapid-evolving heterogeneous hardware makes kernel lock development too slow for modern applications with stringent performance requirements and frequent deployment timelines.

This paper addresses the above issues with application-informed kernel synchronization primitives. We allow application developers to deploy workload-specific and hardware-aware kernel lock policies to boost application performance, resolve pathological usage of kernel locks, and even enable dynamic profiling of locks of interest. To showcase this idea, we design SYNCORD, a framework to modify kernel locks without recompiling or rebooting the kernel. SYNCORD abstracts key behaviors of kernel locks and exposes them as APIs for designing user-defined kernel locks. SYNCORD provides the mechanisms to customize kernel locks safely and correctly from the user space. We design five lock policies specialized for new heterogeneous hardware and specific software requirements. Our evaluation shows that SYNCORD incurs minimal runtime overhead and generates kernel locks with performance comparable to that of the state-of-the-art locks.

1 Introduction

With the ending of Moore's Law and Dennard scaling, the exponential growth of single-processor performance has come to a standstill. Hence, application developers now resort to customization, rather than generalization, to further squeeze out the performance from the hardware. For instance, different applications work best with changing underlying system mechanisms. Although a generic mechanism often provides acceptable performance, it rarely matches the performance of a specialized mechanism, whose performance difference often is an order of magnitude or more [10, 34, 59, 70].

Such a major improvement stems from the fact that specialization bridges the semantic gap between applications and the underlying system [23, 63]: It establishes the *context* under which an application requests functionality from the system. Thus, the underlying system can provide the most suitable implementation or even allow applications to provide their own implementation. The method of specialization is not new. For instance, prior works have targeted the widely used Linux OS that has become a major performance bottleneck for applications [29, 33, 47, 53, 61]. As a result, kernel customization has been extensively studied in the context of scheduling [34], networking [49], storage [70], and accelerators [10]. Although such works mostly focus on the scheduling aspect of the IO, they do not expose one of the basic building blocks of today's software design: concurrency control. Hence, this work takes a step in that direction by enabling the customization of the kernel synchronization primitives that have never been exposed to applications.

Kernel synchronization primitives, especially locks, are of paramount importance to ensuring correctness, achieving good performance, and scalability for applications [8, 9, 35, 37, 52]. Traditionally, kernel developers bake these primitives as a part of the OS implementation. Since it is difficult to change these primitives dynamically, the kernel developers favor supporting common scenarios and make all the decisions regarding their design and implementation. Thus, all these primitives are invisible and are out of reach of applications.

Given evolving hardware and changing software requirements, this static approach of lock design raises two issues: missing hardware and software contexts. From the hardware perspective, applications using kernel components, which rely on such generic primitives, suffer from regression issues in pathological cases [8, 37, 52]. In particular, these primitives

suffer from a high level of contention with increasing core count [8], which requires further optimization for the underlying hardware [11, 21, 51]. In addition, increasing hardware heterogeneity in modern systems further exacerbates this issue [4, 5, 16, 41]. Second, from the software perspective, these baked generic primitives lack application context. As a result, it can lead to pathological cases, such as missing readers-writer context [11], priority inversion [35, 38, 52], scheduler subversion [58], and lock-holder preemption [36].

The current practice of addressing these issues involves developing synchronization primitives for specific scenarios [12, 21, 26, 39, 42, 45, 46, 51]. However, designing, implementing, and verifying new synchronization primitives is challenging. In addition, developers need a huge amount of effort to upstream and maintain them. To satisfy fast-evolving scenarios and requirements, synchronization primitives should be easily changeable and even on the fly instead of providing point-solutions as in previous works.

This paper proposes the idea of *application-informed kernel synchronization primitives* that enables users to develop custom *lock policies* to maximize performance or resolve pathological cases. For example, with an asymmetric multicore processor machine, in which processors operate at a different speed [4, 5, 16], application developers may want to prioritize lock waiters on fast cores to maximize performance. To demonstrate this idea, we design and implement SynCord, a framework built to safely modify kernel locks on the fly without recompiling or rebooting the kernel. We abstract and modularize the semantics of the locking primitives and expose them in the form of APIs. A developer uses these APIs for implementing policies, such as NUMA-awareness, priority boosting, readers-writer preference [11, 21, 37] etc. SynCord then verifies these policies and safely patches the running kernel in the end. It provides the capability to deploy custom code for a wide range of lock instances: from a single lock instance to a set of locks, or every lock in the kernel. Besides deploying lock policies, SynCord further allows users to profile locks at fine granularity. Our approach departs from the conventional tools profiling a fixed set of statistics for all kernel locks [73]. Instead, a user can now collect any lock statistic on arbitrary locks.

The ultimate goal of SynCord is to completely realize the idea of contextual concurrency control [57], which enables users to modify any synchronization primitives from the user space in a safe manner. As a first step in kernel lock customization, our SynCord prototype currently supports non-blocking locks. In particular, SynCord allows users to write their own logic for reordering lock waiters, setting priorities between competing threads to acquire a lock. We support three existing non-blocking primitives: ShflLock [37], CNA [19], and the stock readers-writer lock in Linux. We further demonstrate the generality of SynCord by adapting four different locking algorithms to the kernel and optimizing them based on our evaluation platform. In addition, we provide a case study of lock profiling using SynCord and show how it simplifies the performance analysis of a lock algorithm. Our evaluation shows that the custom algorithms developed with SynCord increase the application performance by up to three orders of magnitude compared to the generic locks.

This paper makes the following contributions:

- **Application-defined concurrency.** We propose the idea of on-the-fly modification of lock design. To realize that, we design and implement the SynCord framework.
- **APIs for non-blocking locks.** We provide a set of APIs that exposes the key decisions of non-blocking locks to implement various lock algorithms.
- **Lock algorithms.** We implement four lock algorithms and optimize them based on the platform. The optimized versions outperform generic locks up to three orders of magnitude.
- **Custom fine-granularity profiling.** SynCord provides custom, fine-granularity lock profiling that simplifies locks' performance analysis with smaller overhead.

2 Background And Motivation

Modifying kernel locking primitives without recompiling and rebooting the OS spans various domains of concurrent OS design. We first discuss the evolution of locks, followed by various mechanisms for kernel customization and the specific need for dynamic patching for kernel locks.

2.1 Lock evolution

Locks are widely used and heavily influenced by hardware. For example, queue-based locks minimize cache-line contention [51] among CPUs by forming a queue of waiters who spin on private cache lines. Hierarchical locks [11, 17, 21] improve application throughput for non-uniform memory access (NUMA) architecture, in which local memory is faster than remote NUMA memory. Such locks exploit the NUMA characteristic by batching requests from the same NUMA node at the cost of higher memory use and lower throughput in non-contended scenarios. CNA [19] and ShflLock [37] address these limitations by dynamically reordering the queue. ShflLock enforces *policies* given by hardware characteristics and software behaviors through shuffling. Although both locks allow designing new lock algorithms by abstracting both hardware and software requirements in the form of policy, their approach is insufficient for changing kernel locks on the fly. A developer still needs to recompile and reboot the kernel to test a new policy. SynCord allows users to develop custom policies and safely deploy them to a live kernel.

2.2 Kernel customization

With the introduction of fast IO devices, hardware accelerators and hundreds of cores, customizing the kernel on the fly is the new norm for improving application performance. However, this idea is not new, as Exokernel [22] is

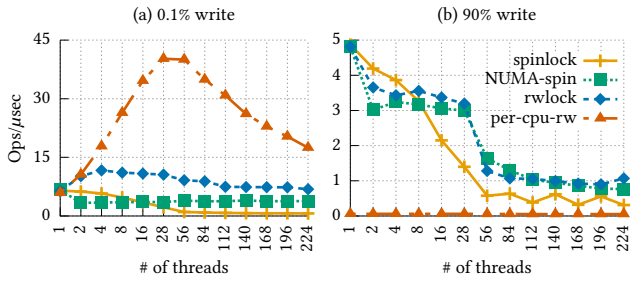


Figure 1: Impact of locks on throughput with different write ratios. The workload is a hashtable benchmark in the kernel [69] where a global lock guards the hashtable.

the first kernel design that enables customization by safely exporting hardware resources to untrusted library OSes and downloading application code to the kernel. Another prominent one that enables customization is the split-level I/O scheduling [70], which enables users to deploy custom I/O scheduling mechanisms across various layers of the storage stack.

Recently, Linux has been allowing user-level applications to customize the kernel by handling page faults in user space [14] or using the eBPF framework [23]. eBPF has seen wide deployment at various places in the kernel. For example, EXTFUSE speeds up user-level file systems with eBPF [6]. eXpress Data Path (XDP) [63] introduces a programmable network data path that allows a user-supplied eBPF program to control network packets. Moreover, recent work proposed delegating kernel operations to user space. For example, Syrup [34], ghOSt [28] and Scheduler BPF [15] allow users to specify scheduling policy and deploy it in the kernel networking stack and thread schedulers. Snap [49] enables the development of networking features in user space, while DPDK [61] and SPDK [29] provide libraries to accelerate packet processing and develop storage features. Compared to these works, SynCord takes a step further in customizing the kernel: it allows users to control the concurrency mechanisms in the underlying kernel.

2.3 Application-defined locking matters

Figure 1 illustrates the fact that one lock design cannot perform the best in all scenarios. For example, when the workload is read-dominant, rwlock outperforms spinlock because spinlock requires mutual exclusion even between read operations. In particular, per-CPU rwlock works better than a centralized one by avoiding cache traffic caused by a reader indicator across cores. However, with a write-dominant workload, the per-CPU rwlock performs the worst. In addition to the application semantics, underlying hardware also requires a different lock [17]. The NUMA-aware spinlock performs better than the MCS spinlock when threads execute across multiple sockets, but MCS can be a better choice on a single socket machine for the first few threads.

One might solve this problem by designing and implementing a special kernel lock. However, developing and maintaining kernel locks customized for each application and hardware is difficult, time-consuming, and costly. Meanwhile, SynCord eases the development of new lock algorithms. First, unlike other subsystems, synchronization primitives are not well isolated in the kernel. Hence, changes to synchronization primitives require understanding the surrounding details that impact a lot of other kernel codes. SynCord provides modularity for synchronization primitives. Second, SynCord APIs serve as an abstraction layer. These APIs hide the underlying tricky implementation details of lock, such as concurrency, memory model, and atomic instructions use. Instead, developers implement policies for scheduling waiters, such as what to do before and after acquiring a lock, and which type of waiters should be prioritized (§5). Moreover, locks designed with SynCord require no changes to the kernel's components and achieve similar speed up with only a few lines of code (Table 5).

2.4 The need for dynamic lock patching

Apart from the difficulty of designing and implementing new lock algorithms in the kernel, installing a modified kernel requires a system reboot. However, there are common scenarios where applications or underlying hardware change during execution, requiring live kernel lock changes. In terms of application changes, applications whose performance matters might change over runtime. This case is possible in a cloud environment, as multiple applications execute in a particular order. In addition to performance, applications try to maintain some form of service level agreements in the form of latency or fairness. Besides this, a scenario of runtime hardware modification is virtual machine (VM) live migration [2]. For example, if a cloud provider migrates a VM from a single socket machine to a multi-socket NUMA machine, users should modify their kernel locks' policies to handle the NUMA behavior. Moreover, with increasing hardware heterogeneity, applications require policies incorporating both hardware and software policies for better performance. One might argue that the traditional approach of kernel patching is sufficient. In this case, the conventional static kernel patching approach cannot efficiently handle such scenarios, motivating the need for the SynCord dynamic approach, which allows developers to implement specific APIs and patch a set of locks, while generally ensuring safety properties.

3 The SynCord Framework

SynCord is a framework for customizing kernel locks on the fly without recompiling or rebooting the kernel. To modify kernel locks, a user writes custom lock algorithms in user space, and SynCord safely deploys them in the kernel. SynCord can patch individual lock instances or every lock in the kernel. In addition, SynCord enables fine-grained profiling of kernel locks, helping users better understand the impact the kernel has on their application. We design

Type	Group	API	Description
Safe	General	① `void lock_to_acquire(lock)`	Invoked before acquiring the lock.
		② `void lock_acquired(lock)`	Invoked after acquiring the lock.
		③ `void lock_to_release(lock)`	Invoked before releasing the lock.
		④ `void lock_released(lock)`	Invoked after releasing the lock.
	Fast path	⑤ `void lock_to_enter_slowpath(lock, node)`	Invoked before entering the slow path.
		⑥ `bool lock_enable_fastpath(lock)`	If true, allow acquiring the lock by the fast path.
	Waiter reordering	⑦ `bool should_reorder(lock, anchor, curr)`	If true, move thread `curr` forward in the queue.
		⑧ `bool skip_reorder(lock, anchor)`	If true, skip the current reordering operation.
Unsafe	Lock bypass	⑨ `bool lock_bypass_acquire(lock)`	If true, bypass lock aquisition.
		⑩ `bool lock_bypass_release(lock)`	If true, bypass lock release.

Table 1: A summary of SynCord APIs. General APIs intercept the entry and exit points of the lock acquire and release phase. Today most of the lock algorithms have at least two paths to enter the critical section: fast path and slow path. Here, the fast path APIs intercept the fast path access to acquire the lock. Meanwhile, the slow path provides waiter reordering APIs that control the reordering of waiters for lock acquisition. Lock bypass APIs allow threads to bypass locks. The lock bypass APIs allow expert developers to design their algorithm, which comes at their own risk.

SynCord to modify kernel locks as a sandbox that adds new policies on top of existing locks.

Design goals. SynCord has three main design goals:

- *Correct lock patching.* SynCord must maintain the mutual exclusion of the lock instances being patched and should not introduce any correctness bugs through the process of patching.
- *Sandboxed user's code.* Users may provide unsafe code that leads to mutual exclusion violation. SynCord aims to prevent such code from corrupting the kernel as long as they use SynCord safe APIs, so that relieves users' concerns about the correctness of their lock design.
- *Usability and expressiveness.* SynCord aims to provide APIs expressive enough to tune kernel locks for various platforms or requirements.

To strike a balance between expressiveness and sandboxed impact, we design two sets of APIs. A set of *safe* APIs (①–⑧ in Table 1) guarantees mutual exclusion for general use, and the other set of *unsafe* APIs (⑨, ⑩) grant expert kernel developers full control of locks at their own risk.

Moreover, we envision that a single organization uses SynCord to modify kernel locking primitives. In particular, the sysadmins of that organization, with root privileges, handle the conflicting policies for various applications contending on the same lock or a set of locking instances. We follow this model because unlike other subsystems, locking primitives guard shared resources, which an unprivileged user should not change. In addition, we assume that such kernel changes do not occur frequently. Thus, a single policy optimized for underlying hardware or applications' usage patterns may last several minutes.

3.1 SynCord overview

Figure 2 illustrates SynCord's key components and workflow. SynCord exposes a set of APIs (Table 1) to abstract underlying lock implementation and allows users to write

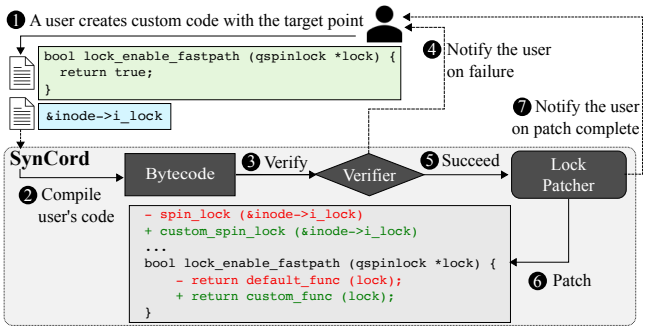


Figure 2: Overview of SynCord's key components and workflow. (1) A user writes custom lock code and specifies lock instances to patch; (2) SynCord compiles the user's lock code (*e.g.*, eBPF) and (3) verifies basic properties of the compiled bytecode. (4) SynCord notifies the user if the verification fails, (5) otherwise loads the program into the kernel and generates a patch. (6) The lock patching module patches the kernel to call compiled bytecode on predefined hook points.

custom kernel locks in the abstracted layer. These APIs are the pre-defined hooking points that developers use for inserting their custom code to control the logic of underlying locks. To use SynCord, a privileged user first specifies the lock instance they want to patch and writes the custom locking code in C with SynCord APIs in a separate file (❶). SynCord processes this file in a semi-interactive fashion. It first reads the file and compiles the custom code (❷). It then passes the compiled program to the verifier that performs static analysis to validate the safety requirements (❸). The verification process usually takes a few milliseconds. If the verification fails, SynCord notifies the user (❹). Otherwise, it loads the policy into the kernel and gets a unique ID of the policy (❺). With the ID, SynCord patches the locking func-

```
def spin_lock(lock):
  lock_to_acquire(lock) # ① Hook the start of lock acquire

  if lock_bypass_acquire(lock): # ⑨ bypass lock acquire
    # lock acquisition is bypassed: used by lock experts
    return

  # If fastpath is enabled, first try to acquire the lock
  # instead of going into the wait queue
  if lock_enable_fastpath(lock) and # ⑥ can steal lock?
                  CAS(&lock.state, UNLOCK, LOCKED):
    lock_acquired(lock) # ② lock is acquired
    return

  node = Node() # A node to join the queue
  lock_to_enter_slowpath(lock, node) # ⑤ Hook before enqueuing
  queued_spin_lock_slowpath(lock, node) # Time to join the queue
  lock_acquired(lock) # ② Hook the start of critical section

def spin_unlock(lock):
  lock_to_release(lock) # ③ Hook the end of critical section

  if lock_bypass_release(lock): # ⑩ bypass lock release
    # lock release is bypassed; used along with ⑨
    return

  lock.state = UNLOCK # Lock released; critical section ends
  lock_released(lock) # ④ Hook right after critical section
```

Figure 3: Pseudocode of `spin_lock` and `spin_unlock` with SynCord APIs. We place the waiter reordering APIs in the slow path of the existing `qspinlock` (`queued_spin_lock_slowpath`) function [13].

tions to execute the policy at pre-defined hooking points (❻). The patching process is time-consuming, as the patching module (`Livepatch`) has to find a quiescence period, *i.e.*, no task is executing the locking functions to patch. This period can last a few seconds. Finally, SynCord notifies the user after the patch completes (❼).

3.2 Programming with SynCord

To design a custom kernel lock with SynCord, a developer first specifies a set of lock instances to patch and implements a new policy by writing code blocks for each API. At a high level, there are two main purposes for developers to write code in the APIs: to enforce user-defined policies on scheduling waiters, and fine-grained profiling. From the scheduling perspective, a lock algorithm ensures the mutual exclusion property while scheduling a set of waiters based on user requirements *e.g.*, FIFO. Thus, SynCord exposes various means to schedule lock waiters, such as queue ordering and backoff schemes. In addition, both custom lock design and profiling often record extra information. Thus, we also provide auxiliary data structures (refer to §3.2.2) that serve as the extra storage space for the custom code.

3.2.1 SynCord APIs

Table 1 summarizes the APIs in the current SynCord prototype. The APIs expose several key behaviors of queue-based non-blocking locks, especially the ordering between lock waiters. We design SynCord APIs to be general across many existing lock designs and safe enough to use in user space. Most locks have well-known interfaces: acquire() and release(). Hence, the first category of our APIs (①–④) hooks these interfaces. Figure 3 shows the pseudo-code of `spin_lock` and `spin_unlock` with the hooking points for SynCord APIs. The *General* APIs allow users to intercept the entry and exit points of the acquire and release phase. These APIs are particularly useful for lock profiling as their hooking points are inspired by Linux's lockstat to profile every kernel lock. For example, users can use these APIs to record the time spent in acquiring the lock or the time spent in the critical section.

The second set of APIs—the *Fast path*—hooks the entry of a slow path (⑤) or controls the fast path of the lock (⑥). The fast path uses test-and-set-based lock for low contention scenarios [13, 19, 37]. For example, in `qspinlock`, CNA and ShflLock, a thread first tries to issue a test-and-set instruction to grab the lock, and only enqueues itself on failure. The fast path optimizes performance but may impact fairness, which now can be easily controlled with APIs. The slow path involves queue maintenance when the lock is in use, which applies to almost all queue-based lock algorithms [13, 44, 54, 55].

To design policies controlling the order to acquire a lock, we rely on a queue-based lock design that provides a powerful abstraction to reorder the waiting queue on the fly without using extra memory. Both CNA and ShflLock allow arbitrary dynamic queue ordering to achieve user-defined policies. Thus, we provide two *waiter reordering* APIs (⑦–⑧). `should_reorder()` moves a waiter in front of the queue by comparing the current node with an anchor node. `skip_reorder()` skips the reordering procedure. When the reordering is skipped, the waiting queue goes back to the first-in-first-out policy to maintain waiting threads. This is useful to enforce fairness for specific scenarios or purposes.

Although these APIs can implement several queue-based lock algorithms, they cannot change the basic mechanism of the underlying kernel locks. However, an experienced lock developer may want to redesign the kernel lock completely [17, 20, 21]. Thus, SynCord introduces a set of APIs that allow the custom code to bypass the lock acquire and release phase (⑨–⑩). These APIs grant users complete control of the kernel lock and thus allow users to design arbitrary lock algorithms. Since these APIs bypass underlying locks, it is the user's responsibility to correctly maintain the mutual exclusion property.

A point to note is that most of the APIs only introduce performance bugs with incorrect usage. Although the reordering API might affect the fairness, SynCord prevents threads from starvation with runtime checks. Meanwhile, our APIs are designed not to introduce correctness bugs (*e.g.*, infinite loops, mutual exclusion violations) except for the lock bypass APIs.

3.2.2 Auxiliary data structures

Sometimes, designing new lock algorithms or profiling existing locks might require extra information. For example, to implement a NUMA-aware lock scheduling algorithm (§5.1), each waiter needs to record its socket ID. Hence, we support auxiliary data structures to save some semantic information. In particular, we support three such types: per-node data, per-lock data, and global data. Per-node data is associated with a thread (node) that waits to acquire the lock. Its lifetime starts when a thread joins the waiting queue and ends when the thread acquires the lock. Per-lock data is associated with a lock instance, and global data plays the identical role as global variables in the kernel. Both per-lock and global data structures are created and destroyed explicitly by SynCord and follow the lifetime of the associated policy in most cases. The user, while implementing custom code, also defines the required types of auxiliary data structures, which get compiled along with the lock.

3.3 SynCord properties for lock design

Unlike current kernel customization approaches that localize resources within a process model, exposing dynamic lock modification requires reasoning about the mutual exclusion properties, minimizing the impact of starvation and ensuring that we patch the lock code correctly.

Sandboxed impact. If the user provides buggy code, SynCord should prevent such code from corrupting the kernel. SynCord guarantees code safety: memory safety (no access to illegal memory address), termination (no infinite loop), liveness (no deadlock) and mutual exclusion. The current verifier, which relies on the eBPF verifier, uses static analysis to enforce code safety. In particular, SynCord APIs pass a lock instance as a read-only argument to prevent arbitrary changes to the lock state during the lock acquisition and release phases. For instance, SynCord APIs (except the bypass ones) do not modify the functioning of the existing lock algorithm, such as atomic instruction, barriers, and concurrent executions. Because of this, SynCord does not introduce any new deadlock situation, meanwhile maintaining the liveness of the underlying lock even after adding the user logic. Our APIs only provide suggestions/hints to existing locks, as we do not change their underlying working. Hence, it is impossible to incur mutual exclusion violation. Meanwhile, we advise only lock experts to use the bypass APIs for complete access to the lock state.

Avoiding starvation. The reordering of waiters by SynCord introduces starvation that can severely affect the kernel response. We address this issue with bounded runtime checks. For example, if a custom lock uses backoff, we ensure that a waiter only waits for a maximum amount of time. To ensure this behavior, SynCord disables the custom logic for the respective APIs if the thread is suffering from starvation. We currently set the bounded time to 10 ms.

Correct lock patching. If a user provides the correct code, SynCord must address three issues: 1) only patch the required code, 2) apply the patch when the system is in a quiescence state to avoid any inconsistency, and 3) resolve the lock patch if multiple conflicting policies exist. We address these issues by using the mature patching service in Linux: `Livepatch` [32], which works as follows. `Livepatch` first generates a difference between the changed code. It then compiles the diff as a kernel module. Now, `Livepatch` inserts the module once the system reaches a quiescent state, *i.e.*, when a thread leaves the kernel space or CPUs are in idle mode.

In the case of SynCord, `Livepatch` can fail to insert the code if a user patches the same lock instances with multiple policies. We address this issue as follows: Before applying a patch, SynCord checks for an existing patch on the lock. If so, it aborts and reports the conflict to the system administrator. We also support re-patching the same locking call site by bypassing the previous check. Thus, SynCord provides the flexibility to resolve patch conflicts. For example, the user can completely override the old patch with the new one. Alternatively, they can develop and apply a new patch by manually merging those two patches.

4 SynCord Implementation

We now discuss the implementation of SynCord. First we present a summary of SynCord's implementation and then the two existing mature Linux kernel tools SynCord builds upon: eBPF (§4.1) and `Livepatch` (§4.2).

The current SynCord prototype targets non-blocking locks and supports both exclusive locks (*e.g.*, spinlocks) and readers-writer locks. For spinlocks, we implemented SynCord with the stock Linux spinlock, CNA [19] and ShflLock [37]. For readers-writer locks, we implemented SynCord with the readers-writer locks in ShflLock and the Linux kernel. Our current prototype uses Linux v5.4.

SynCord requires a one-time kernel modification to expose APIs (§3.2.1), extra eBPF helper functions, and runtime checks. Exposing the current SynCord APIs is relatively straightforward. Except for the waiter reordering APIs, we exposed all the other APIs by inserting a dummy function. We expose the waiter reordering for ShflLock and CNA in the form of a comparison function and also support the case for skipping the comparison with the `skip_reorder()` function. In total, we modify 143 lines of code in the Linux kernel. Our code is publicly available at https://github.com/rs3lab/SynCord.

4.1 eBPF for SynCord

eBPF allows applications to run custom code at specific points in the kernel (called hook points or target points). To use eBPF, a user first writes a program in C and compiles it into the eBPF bytecode. The kernel then loads the bytecode, verifies the memory safety, and then deploys it at the specified hook points. The eBPF verifier uses static analysis to check any illegal memory access in the program and also

verifies if the program terminates. To guarantee the termination of a program, the verifier only allows bounded loops [65] and rejects unreachable instructions or out-of-bound jumps. eBPF further tries to ensure safety by only whitelisting a set of safe functions (called helper functions), so that applications can obtain system state, such as the current time, CPU ID, etc. SynCord exploits the eBPF safety guarantees to enforce several safety requirements (§3.3).

Several lock algorithms require threads to spin until they satisfy a set of specific conditions. However, the eBPF verifier does not allow loops since it cannot guarantee termination. To address this issue, SynCord introduces a new eBPF helper function: `backoff()`. With this helper function, a thread terminates its spinning either by meeting the defined condition or the specified time is over. SynCord further sets an upper limit on the timeout value (10 ms) to avoid starvation. We designed `backoff()` as an eBPF helper function so users can use these functions in any SynCord APIs.

4.2 Kernel livepatching for SynCord

Kernel livepatch [32, 56, 60, 66] modifies the kernel on the fly without rebooting the system. While eBPF alone can deploy user-defined code into the kernel (§4.1), the effect is global and thus affects all the lock instances in the kernel. To support a finer deployment granularity (§3) and auxiliary data structures (§3.2.2), SynCord uses `Livepatch`, specifically `Kpatch` [60] to deploy the custom code.

Implementing auxiliary data structures. We expose three types of auxiliary data structures: per-node data, per-lock data, and global data. Per-node data stores additional information when a thread joins a waiting queue in a queue-based lock design. Currently, the per-node data is 16 bytes but it is aligned at a cache-line boundary (64 bytes) to avoid false sharing. Thus, SynCord uses the remaining 48 bytes to store extra information. Presently, the current spinlock size is fixed at 4 bytes, and modifying the lock itself increases the memory footprint of any lock instance. Hence, for per-lock data, we use shadow variables [40] to allocate extra memory only for the target lock instances. In particular, we store the auxiliary data inside an in-kernel key-value store created by Livepatch. The address of a target lock instance serves as the key, while the value is per-lock auxiliary data. In addition to the per-lock data, global data such as per-CPU data can be also stored in that key-value store. SynCord frees the extra memory allocated as shadow variables when it removes the corresponding policy. In other words, SynCord does not modify the structure of the lock itself, instead it stores the additional per-lock data separately from the parent lock object. Hence, our design choice does not increase the memory footprint of all locks in the kernel. SynCord only allocates memory for the target locks, thereby having no additional memory footprint without an installed policy.

Workload	Lock: Usage
MWRL [52]	**rename_lock:** Rename files within a directory
lock1 [7]	**files_struct.file_lock:** fcntl and fd allocation
page_fault1 [7]	**mmap_sem:** Anonymous memory page-fault
LevelDB [25]	**futex** contention on futex hash bucket
Metis (wrmem) [48]	reader side of **mmap_sem** on page-fault
SCL-Victim [58]	**rename_lock:** Rename files from/to an empty directory
SCL-Bully [58]	**rename_lock:** Rename files from an empty directory to a directory with 1M files

Table 2: Lock usage in each benchmark.

```
# per-node auxiliary data structures
class node:
  ...
+ int socket_id # Store socket ID for the thread

def lock_enable_fastpath(lock): # Allow lock stealing
  return True

def lock_to_enter_slowpath(lock, node):
  node.socket_id = get_numa_id() # Record socket ID for waiter

# Return true if anchor and curr are in the same socket
# Applicable to both ShflLock and CNA
def should_reorder(lock, anchor, curr):
  return anchor.socket_id == curr.socket_id

# randomly skip reordering to pass the lock to another socket
def skip_reorder(lock, anchor):
  return random() & 0xffff
```

Figure 4: Pseudocode of a NUMA-aware lock with SynCord.

5 Use Cases

We discuss the design and evaluation of various use cases enabled by SynCord. We first cover lock scheduling algorithms that manipulate the ordering of the lock acquisition. SynCord provides the means to define a custom lock acquisition order either by reordering the waiting queue or by blocking specific threads from joining the queue. We classify the lock scheduling algorithms into two types: (1) acquisition-aware scheduling, which considers the characteristics of lock waiters to enter the critical section, such as NUMA-awareness (§5.1) or biased readers-writer (§5.4); (2) occupancy-aware scheduling, which involves scheduling based on the time a thread spends in the critical section (§5.2, §5.3). The second use case focuses on customized fine-grained profiling (§5.5). Finally, we discuss our experience of using SynCord to implement these use cases (§5.6). Note that every lock implemented with SynCord is marked with "⋆" in figures. "-static" represents Linux kernel compiled with a static implementation of the equivalent locking strategy.

Experimental setup. We evaluate SynCord on an 8-socket, 224-core machine equipped with Intel Xeon Platinum 8276L CPUs. The machine runs Ubuntu 20.04 with Linux kernel 5.4.0 with disabled hyperthreading. Table 2 lists the benchmarks we use for the evaluation.

5.1 NUMA-aware spinlock

Motivation. Modern servers have non-uniform memory access (NUMA) architectures, with multiple sockets, multi-

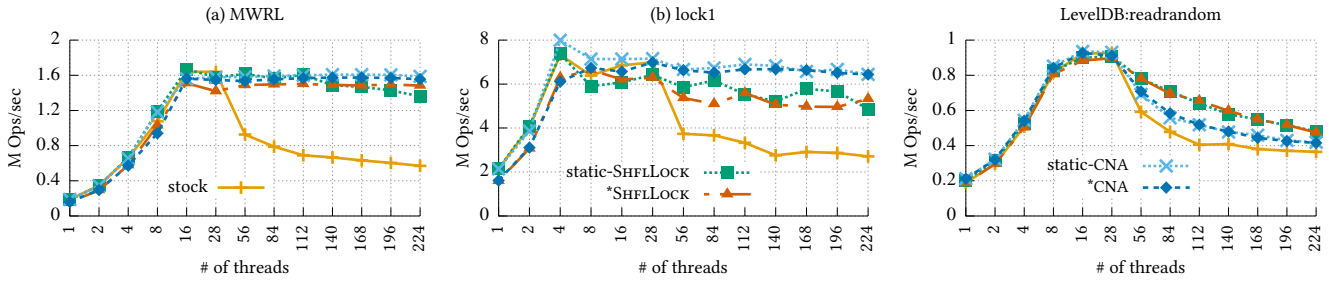


Figure 5: Comparison between kernel locks (stock, ShflLock, CNA) and their SynCord equivalent for NUMA-aware scenario.

ple cores, locally attached memories, and shared last-level caches. In such a server, accessing local memory is faster than accessing remote NUMA memory [21, 62]. NUMA-aware locks improve application throughput by batching lock acquisitions from the same socket together [19, 37].

Design. SynCord adopts the dynamic reordering mechanism for NUMA-aware locking from ShflLock and CNA, and batches requests from the same socket. NUMA locks ensure long-term fairness by periodically passing the lock to another socket. Figure 4 shows the implementation of this algorithm with SynCord. We add a per-node auxiliary integer: `socket_id` to record the socket ID of each waiting thread (line 4). The pseudocode denotes it inside `Class`, while the actual implementation is the `struct` of C code. We also enabled the fast path (lines 6–7) for lock stealing. If the fast path fails, the waiter enters the slow path after recording the socket ID (lines 9–10). The reordering process occurs in the slow path, and the underlying lock decides a reordering strategy. For example, in the case of ShflLock, the shuffler (`S`) first checks whether it should skip reordering by invoking the `skip_reorder` (lines 18–19) API. If not, `S` iterates the queue starting from itself; it invokes `should_reorder` (lines 14–15) with itself being the `anchor` and a waiting thread as `curr`. For all the waiting threads that make `should_reorder` return true, `S` moves that waiter forward, which groups waiters from the same socket. If `skip_reorder` returns true, `S` assigns the next waiter as the new shuffler, which ensures long-term fairness. The CNA lock also applies the same approach of grouping, but with a queue-splitting mechanism.

Evaluation. Figure 5 compares the stock version and two versions of ShflLock and CNA. The first is a static implementation requiring kernel re-install and reboot, while the other is the SynCord-based (marked $*$). Since ShflLock and CNA have a NUMA-aware policy on default, the evaluation shows how much overhead is introduced by SynCord's dynamic approach compared to the static implementation of the identical policy. We evaluate two microbenchmarks that contend on a single lock, and LevelDB's readrandom, which contends on the lock guarding the `futex` bucket. We find that the SynCord-based locks enforce a NUMA-aware policy on the fly without any significant overhead. Their performance is similar to their static counterparts and outperforms the stock version present in the Linux kernel, without having to compile or reboot the kernel. LevelDB's performance drops can be attributed to its use of a global database lock, which is not NUMA-aware.

5.2 Asymmetric multicore lock

Motivation. Asymmetric multicore processors (AMP) [4, 5, 16] consist of heterogeneous cores with different computing powers: energy-efficient slow cores and power-hungry fast cores. By combining both fast and slow cores in one processor, an AMP machine can adjust for dynamic usage patterns, for example, utilizing all cores to maximize the performance or using only slow cores for better energy-efficiency. Moreover, an AMP-aware scheduler can place low computation tasks on slow cores and place compute-intensive tasks on the fast ones. Unfortunately, current lock designs are unsuitable for the AMP architecture, as most lock designs assume homogeneous cores [41]. In particular, their performance significantly degrades when running on an AMP machine. This happens because slow cores execute critical sections up to 4× slower than faster cores [4], leading to lower throughput and higher latency [41]. Moreover, no lock design works efficiently for a multi-socket NUMA server, each has AMP.

Design. Our design is inspired by the LibASL [41], an AMP-aware user space lock without NUMA-awareness. Taking a step further, we extend LibASL's design for future AMP NUMA machines. LibASL works as follows: During a low contention scenario, it allows both slow cores and fast cores to acquire the lock to maximize performance. Meanwhile, during high contention, it penalizes slow cores so that fast cores can acquire the lock more aggressively for better performance. We slightly depart from LibASL with regards to penalty. In particular, we penalize slow core threads by forcing them to wait for a maximum of fixed time (10ms) before acquiring the lock. Providing appropriate wait time prevents starvation and ensures acceptable latency for workloads running on slow cores.

Figure 6 shows our version of the AMP-aware lock implemented using SynCord APIs. We use `MAX_WAIT_TIME` (line 2) as the maximum wait time for threads running on slow cores, and per-node socket ID (line 6) is used for NUMA-awareness.

```
1  class global_aux: # Global auxilary data structure
2  + int MAX_WAIT_TIME = 10ms # Max backoff for the slow core
3
4  class node: # Per-node auxiliary data structures
5    ...
6  + int socket_id # Store socket ID for the thread
7
8  def lock_enable_fastpath(lock): # Allow lock stealing
9    return True
10
11 def is_lock_unlocked(lock):
12   return lock.val == UNLOCK # Check if the lock is unlocked
13
14 def lock_to_enter_slowpath(lock, node):
15   node.socket_id = get_numa_id() # Record socket ID for waiter
16   cpu = get_processor_id() # Get the CPU ID of the thread
17
18   # Fast core thread directly enters; the slow one joins if:
19   #   1. the lock is not held or
20   #   2. it has been waiting for a predefined time
21   if is_slow_core(cpu):
22     backoff(lock, MAX_WAIT_TIME, is_lock_unlocked)
23
24 # Group same socket thread together
25 def should_reorder(lock, anchor, curr):
26   return anchor.socket_id == curr.socket_id
27
28 def skip_reorder(lock, anchor):
29   return rand() & 0xffff # Skip reordering to avoid starvation
```

Figure 6: AMP algorithm pseudocode with SynCord.

Similar to ShflLock, we allow lock stealing in the fast path (lines 8–9). On failing the fast path, the waiter assigns itself a socket ID (line 15) and checks its core type (line 21). If the waiter runs on a fast core, it immediately enters the slow path and joins the waiting queue. Otherwise, the waiter needs to wait before joining the waiting queue using the `backoff` function (lines 21-22). The thread on the slow core spins either until the `MAX_WAIT_TIME` has elapsed or if the lock has been released (lines 11–12). After returning from the `backoff` function, the waiter finally enters the slow path and joins the waiting queue. Once a thread enters the slow path, it follows a similar strategy as that of the NUMA lock for queue reordering and skipping. Thus, our NUMA-aware AMP lock prioritizes threads on fast cores before joining the queue, and further prefers the thread from the same socket after entering the queue. On the other hand, slow cores do acquire the lock after spinning for a predefined time (10ms). With the time-bound spinning, our approach prevents starvation while boosting application throughput.

Evaluation. Since there is no NUMA-AMP machine, we emulate the AMP environment by changing the CPU frequency. The fast cores are 4× faster than the slow cores and each socket has 14 fast and 14 slow cores, respectively. We use the same workloads as before (§5.1) and compare stock, ShflLock, and AMP. AMP is implemented statically (static-AMP) and using SynCord (*AMP) to compare the overhead coming from SynCord. Figure 7 shows that AMP outperforms both ShflLock and stock by 1.5x and 13.4x each and maintains the performance with increasing core count. To dig deeper, we measure the throughput of fast and slow cores separately. We find that all three locks have similar throughput under low contention (eight threads) for all workloads. This happens because all three locks are able to steal locks during the fast path. However, when the contention becomes high, both stock and ShflLock allow slow cores to acquire the lock, leading to lower throughput. On the other hand, AMP lets fast cores acquire the lock most of the time and thus achieves higher throughput.

5.3 Scheduler-cooperative lock

Motivation. Patel *et al.* [58] described the *scheduler subversion* problem, in which competitive threads hold the same lock for varying times, leading to a subversion of CPU scheduling goals. In particular, current CPU schedulers let each thread have an equal share of the CPU time. Suppose two threads are spending most of their CPU time executing in a critical section protected by the same lock; one thread (the bully thread) holds the lock for a much longer time (*e.g.*, orders of magnitude longer) than the other (the victim thread). In this case, the bully thread essentially receives a much longer CPU time than the victim thread, subverting the scheduling goal. This can lead to pathological cases of denial of service attacks, and lower application performance [58].

Design. To address this problem, we implement a new lock algorithm, that strives to achieve fair hold time across threads: SCL. SCL utilizes ShflLock as the underlying kernel lock and uses SynCord APIs. The algorithm assumes that all threads have the same priority and receive the same CPU time. The algorithm tracks the time spent in the critical section for each thread. If one thread holds the lock longer than its share, it cannot acquire the lock until other threads have received an equal chance to acquire the lock. On top of SCL, we also implement a NUMA-aware version: NUMA-SCL.

Figure 8 presents the implementation of SCL. We have not included the NUMA part for brevity, which is similar to §5.1. We introduce several auxiliary data structures to implement this algorithm: a per-thread lock hold time variable (line 3), a per-thread variable for recording the beginning of the critical section (line 5), a per-lock integer for counting contending thread (line 9), and total hold time for each lock (line 10). The algorithm works as follows: Before a thread (`t`) joins the waiting queue, it first computes the lock quota based on the number of threads and overall lock holding time (line 14). Based on the time `t` spent in the critical section (line 15), it waits until other threads get the equal opportunity (lines 15–20) by backing off for that approximated time (line 20). We track the per-lock total lock hold time and per-thread lock hold time by tracking when the thread enters the critical section (line 23) and update the overall lock usage in the release phase (lines 26–31).

Evaluation. We evaluate five locks with a workload proposed by Patel *et al.* [58]. The workload creates two types of threads: victim threads and bully threads that contend on the `rename_lock`. Table 2 shows the configuration of the work-

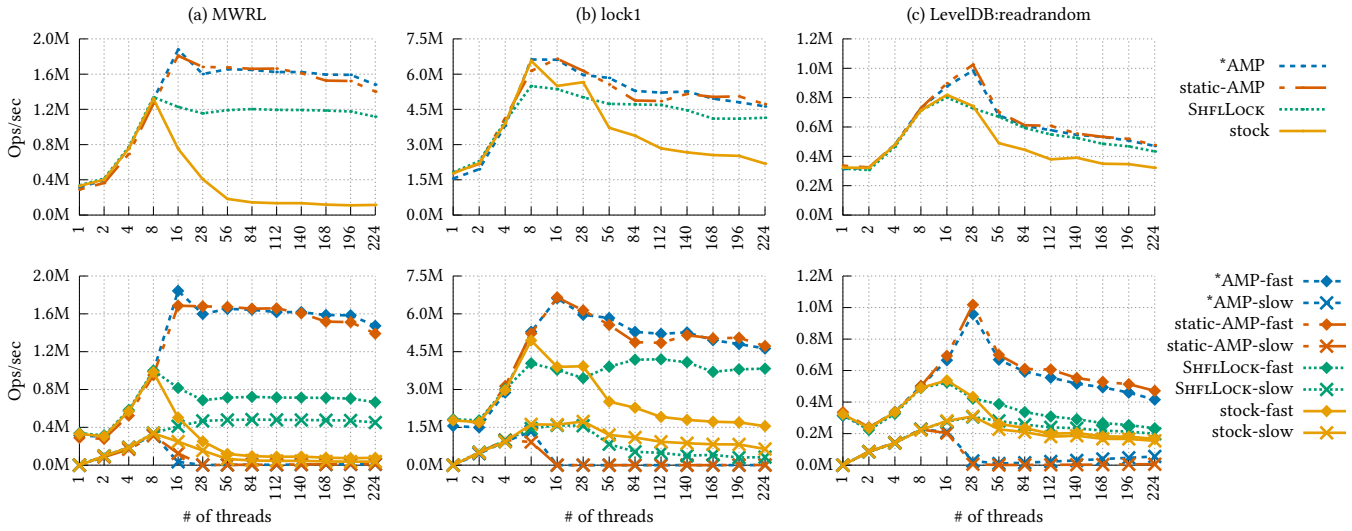


Figure 7: Overall throughput (up) and the throughput of fast and slow cores (down) for stock, ShflLock, and AMP implemented statically (static-AMP) and with SynCord (*AMP). Throughput of AMP-slow after 28 threads is very low but not zero. Refer to Table 4 for details.

```
class global_aux: # Global auxiliary data structures
    # Per-thread variable to record the lock hold time
+   int lock_hold_time<thread>
    # Per-thread variable to timestamp the beginning of CS
+   int cs_beg_ts<thread>

class lock: # Per-lock auxiliary data structures
    ...
+   int num_threads # Threads contending for the lock
+   int tot_lock_hold_time # Total lock hold time of all threads

def lock_to_enter_slowpath(lock, node):
    # Calculate the lock hold quota
    quota = lock.tot_lock_hold_time / lock.num_threads
    if lock_hold_time[curr_thread] > quota:
        # Exceeded local quota. Wait until threads
        # use same amount of quota
        wait = (lock_hold_time[curr_thread] * lock.num_threads)
        wait -= lock.tot_lock_hold_time
        backoff(lock, wait, None)

def lock_acquired(lock):
    cs_beg_ts[curr_thread] = get_time() # get timestamp after acq

def lock_to_release(lock):
    # Calculate the length of the critical section
    cs_len = get_time() - cs_beg_ts[curr_thread]

    # Update the lock usage for this thread and the lock
    lock_hold_time[curr_thread] += cs_len
    lock.tot_lock_hold_time += cs_len
```

Figure 8: Pseudocode of NUMA-SCL with SynCord. We omit the NUMA-awareness code (refer to Figure 4).

load, in which the bully holds the lock up to three orders of higher magnitude than the victim. These five locks include Linux's stock, ShflLock, static-SCL [58], *SCL (SCL without NUMA) and *NUMA-SCL. We implement the last two SCL locks with SynCord. Figure 9 shows the overall throughput and Jain's fairness index [31] of these locks. Jain's fairness

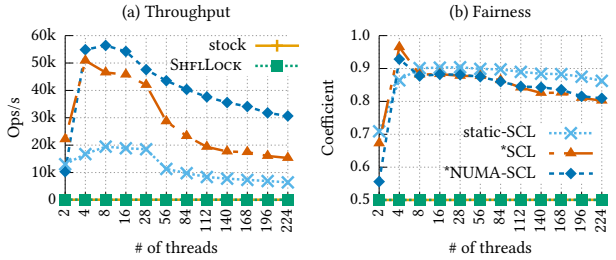


Figure 9: Impact of different lock designs on throughput and fairness. The workload is a rename program contending on rename lock where bully threads hold lock much longer than victim threads. Refer to Table 2 for more details.

index ranges from zero to one, where zero and one indicate completely unfair and fair, respectively. Since the SCL policy ensures that each thread holds the lock for the same length, all SCL versions allow the victim threads to hold the lock much more often than the bully threads. As a result, SCL implementations achieve orders of magnitudes higher throughput than both stock and ShflLock. Moreover, *NUMA-SCL outperforms the non-NUMA version (*SCL) by minimizing cache-line bouncing, thereby having the best overall throughput. The static implementation of SCL performs worse than SynCord versions, as it requires periodic scanning of the thread lists to remove inactive waiters. Such an approach is not required for SCL locks with SynCord because the user can dynamically decide the timeframe to enforce the lock hold time fairness.

We further confirm the aggregated lock hold time of bullies and victims. Figure 10 shows that a bully thread holds the lock

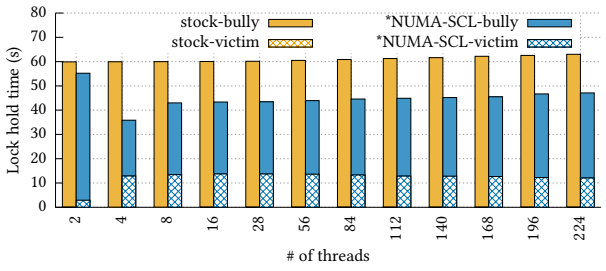


Figure 10: Total lock hold time of bully (B) and victim (V) threads.

for more than 99.9% in the stock version while *NUMA-SCL ensures a better share between bully and victim.

5.4 Biased per-cpu readers-writer lock

This section shows the use of SynCord lock bypass APIs.

Motivation. Readers-writer lock (`rwlock`) is one of the most widely used primitives in Linux [37]. This primitive allows either multiple readers or one writer to acquire a lock. Most `rwlock` designs track active readers with a centralized readers indicator. However, the centralized readers indicator has poor scalability (Figure 1) because frequent atomic instructions for readers result in cache-line invalidation and coherence traffic. Prior work addressed this issue using distributed counters, but with the cost of high memory usage and longer writer latency. Thus, it is a good candidate only for read-intensive workloads [18, 42].

Design. We design a distributed readers-writer lock using the SynCord bypass APIs. Our design is inspired by the `BRAVO` design [18]. With SynCord's dynamic approach, users can enable the distributed `rwlock` only when needed and disable it to avoid unnecessary overhead, such as memory footprint and writer latency. Note that SynCord cannot ensure the mutual exclusion property for the unsafe APIs, and it is up to the lock developers to ensure the correctness. Moreover, we also use another unsafe function (`backoff_unsafe`) that waits indefinitely until the condition is met. We forbid the user from using this unsafe function with our safe APIs, as we throw an error on detecting it.

Figure 11 shows the per-CPU `rwlock` implementation in SynCord. We add two more fields per lock: `rbias` to track the read bias mode, and `visible_readers` table with cache-line aligned entries. Before a reader (`R`) acquires a lock, it first checks the read-biased mode. If the read-biased mode is set, `R` marks itself as an active reader (lines 7–9) and checks the `rbias` again due to a possible race from the writer's side (line 33). If `rbias` is still set, `R` bypasses the underlying lock, else it falls back to the underlying implementation (lines 11–13). At the time of release, `R` checks whether it acquired the lock in the read-biased mode (lines 16–20), and bypasses the underlying lock release if so. `R` is also responsible for setting `rbias` once it acquires the underlying lock without

```
1  class lock: # Per-lock auxiliary data structures
2    ...
3  + int rbias # When set, the lock is on readers-biased mode
4  + int visible_readers[max_cpu] # Distributed read indicator
5  # === Reader ===
6  def lock_bypass_acquire(lock):
7    if lock.aux.rbias: # If in read biased
8      lock.aux.visible_readers[cpu] = 1 # Mark the reader
9      if lock.aux.rbias: # No writer is waiting
10       return True
11     else: # Writer is present
12       lock.aux.visible_readers[cpu] = 0
13   return False
14
15 def lock_bypass_release(lock):
16   # Bypass the lock if reader acquired in rbias mode
17   if lock.aux.visible_readers[cpu] == 1:
18         lock.aux.visible_readers[cpu] = 0
19       return True
20   return False
21
22 def lock_acquired(lock):
23   # Enter the read-biased mode
24   if not lock.aux.rbias:
25     lock.aux.rbias = True
26
27 # === Writer ===
28 def wait_for_reader(lock, cpu):
29   return lock.aux.visible_readers[cpu] == 0
30
31 def lock_acquired(lock):
32   if lock.aux.rbias:
33     lock.aux.rbias = False # Revoke bias
34     for i in range(0, NUM_CPU): # Wait for readers to leave
35       backoff_unsafe(lock, wait_for_reader(i))
```

Figure 11: Pseudocode of the `BRAVO` algorithm. We omit the code to get CPU ID for brevity.

bypassing it (lines 24-25) so that other readers can directly acquire the read lock by setting their respective indicators. In the case of writer acquisition, a writer (`W`) first acquires the underlying writer lock. `W` will only acquire the lock when there are no active readers that hold the underlying lock. After that, `W` further checks for the read-biased mode (line 32). If it is active, `W` first disables it and waits for all readers to exit the critical section that used the per-CPU indicator (lines 34–35).

Evaluation. Figure 12 compares the stock `rwlock` with our version using SynCord (*per-CPU `rwlock`). We evaluate these locks with a `page_fault1` microbenchmark from `will-it-scale` [7] and Metis [48], a MapReduce framework, contending on the reader side of `mmap_sem`. Figure 12 shows that *per-CPU `rwlock` outperforms stock by 2.2x and maintains the performance with increasing core count.

5.5 Dynamic lock profiling

Motivation. Several works [8, 19, 35, 37, 52] have shown that kernel locks mostly determine the scalability of applications. Hence, lock profiling tools are critical to understanding a lock's performance. Unfortunately, only few tools exist for kernel lock profiling, and even those have limited analysis capability. For example, developers often use Linux perf [1] to measure the aggregated CPU cycles in each code region. While this is useful to find lock contention, it does not pro-

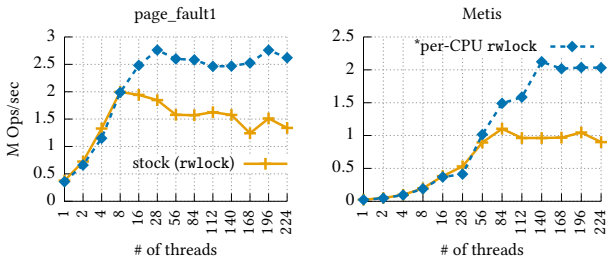


Figure 12: Comparison between Linux `rwlock` and our distributed per-CPU `rwlock` with SynCord. Refer to Table 2 for workload details.

policy	acquisitions	x-socket	avg-batch	violation
SCL+Reorder$_{Bully}$	4042	277	14.59	1659
SCL+Backoff$_{Bully}$	918765	8678	105.87	538

Table 3: Statistics collected by SynCord to analyze our two different implementations of NUMA-SCL: SCL+Reorder$_{Bully}$ and SCL+Backoff$_{Bully}$. We collect this result from the same benchmark used in §5.3 with 224 threads.

vide any lock-specific performance stats, such as the time spent in the critical section. As a result, Linux perf is not always the right tool for understanding lock performance. On the other hand, Linux provides another tool: `lockstat` [73] which exposes various statistics of kernel locks. However, it profiles all the kernel locks together and only shows the system-wide statistics. Moreover, a user can neither choose the lock instance nor specific lock data to profile. To make matter worse, `lockstat` requires a kernel to be compiled with a specific configuration, which significantly increases the size of every lock data structure and introduce performance overhead. For example, a kernel with `lockstat` uses 423MB of extra memory over the stock version even from the booting.

Design. SynCord can patch locks at various granularities, from an individual lock instance to a set of locks. In addition, SynCord provides the ability to profile any lock instance with arbitrary lock-specific performance stats. In particular, a user can now customize which set of lock instances to profile with specific statistics (even the algorithm-specific ones). For example, a user can profile only the rename lock and count the number of lock acquisitions across socket instead of collecting a set of statistics for all the locks in the kernel. We reproduce the statistics provided by `lockstat` using SynCord. Since the hooking points of SynCord General APIs (Table 1) have the same context as those of lockstat, the implementation is straightforward with simple updates of the counters and timestamp in each API.

Evaluation. We compare the overhead of lock profiling between `lockstat` and SynCord. `lockstat` keeps track of 10 counters for each lock. We implement two versions of SynCord-based `lockstat` having identical 10 counters and

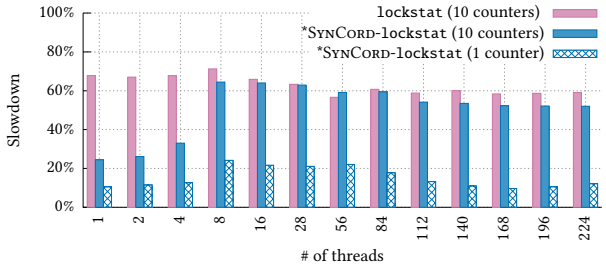


Figure 13: Performance overhead of lock profiling in MWRL for both `lockstat` and our custom implementation of `lockstat` with SynCord. The overhead increases with increasing counters.

policy	# threads	acqst-fast	acqst-slow
AMP+Reorder$_{Fast}$	8	20,340,943	8,192,987
AMP+Backoff$_{Slow}$	8	24,825,644	5,267,531
AMP+Reorder$_{Fast}$	224	571,712	571,354
AMP+Backoff$_{Slow}$	224	37,937,529	54,941

Table 4: Statistics collected by SynCord to analyze our two different implementations of AMP: AMP+Reorder$_{Fast}$ and AMP+Backoff$_{Slow}$. We collect the result for MWRL with the same environment as in §5.2.

one counter. Figure 13 shows that `lockstat` constantly incurs a 60% application slowdown. We observe that the overhead of SynCord profiling increases with an increased number of counters. With 10 counters, SynCord profiler incurs a similar overhead as that of conventional `lockstat`, while it is only 24% for one counter. Furthermore, unlike `lockstat`, SynCord opens the door for lock profiling even on production servers, as SynCord can dynamically turn on the profiling feature and does not introduce overhead when the profiling is uninstalled.

Simplifying performance analysis. We now illustrate an example of how SynCord's custom profiling can significantly simplify the performance analysis. Our initial implementation of the AMP (§5.2) and SCL (§5.3) algorithms only utilized the reordering mechanism of CNA and ShflLock (§5.1). In particular, we only used the `should_reorder` API to enforce both NUMA-awareness to handle hardware characteristics and SCL and AMP algorithms for software requirements. However, the reordering API alone has limitations in strictly enforcing the policy, thus it is difficult to have desired performance as the number of threads increases.

To understand the NUMA-SCL algorithm, we collect the following statistics for the reordering-based algorithm (SCL+Reorder$_{Bully}$), and the current algorithm-based on bully backoff with NUMA-aware reordering (SCL+Backoff$_{Bully}$): the total number of lock acquisitions, the number of lock acquisitions across socket, the average times of lock passing within a socket, and the number of policy violations (a bully

thread acquires the lock when it shouldn't). Table 3 shows that SCL+Reorder$_{Bully}$ fails to enforce the policy, leading to a high violation count. This happens because the reordering approach guarantees that a waiter will acquire the lock once it is part of the waiting queue. Thus, a bully can still grab the lock even though it is penalized, if a victim has not yet joined the waiting queue. Instead, the backoff-based approach (SCL+Backoff$_{Bully}$) avoids this problem by preventing the bully thread from joining the wait queue, thereby effectively enforcing the policy.

Table 4 shows the collected statistics for AMP locks using two variations: AMP+Reorder$_{Fast}$ which uses reordering only to enforce both fast-core preference and same-socket preferences, and AMP+Backoff$_{Slow}$ shown in §5.2. We collect the number of lock acquisitions separately on the fast and slow cores. Under low contention (eight threads), both algorithms have similar throughput. However, under a highly contended scenario (224 threads), the reordering approach cannot correctly enforce the fast-core preference anymore, leading to a performance drop. The same reasoning from SCL holds true here too.

5.6 Experience with SynCord

Policy	LoC	Time
NUMA-aware spinlocks (§5.1)	6	3 hours
Scheduler-cooperative locks (§5.3)	30	18 hours
Asymmetric multicore-aware locks (§5.2)	15	8 hours
Scalable reader-writer locks (§5.4)	36	5 hours
Lock profiling (§5.5)	36	1 hour

Table 5: Development effort of the use cases.

We now discuss the efforts and lessons we have learned in developing the use cases with SynCord.

Lock development effort. Table 5 summarizes the development effort of all use cases. We spent most of the development time understanding, debugging, and testing the lock algorithm. Implementing the algorithms in SynCord involves only tens of lines of code and is relatively straightforward. In addition, SynCord allows users to modify the lock without kernel installation and rebooting, which dramatically reduces the overall development effort.

Hardware is (still) the key factor of performance. We include the NUMA grouping policy even for SCL and AMP algorithms to make them perform well on a NUMA machine. Initially we did not plan to include the NUMA grouping algorithm since we thought the performance gain achieved by task-specific scheduling should dominate. For example, due to the big performance gap between fast and slow cores in AMP machines, scheduling fast cores should achieve good enough performance without NUMA grouping. However, this has proven to not be the case. LibASL performs even worse than ShflLock on our emulated AMP NUMA machine due to the cache-line bouncing among sockets. Hence, we believe that a lock developer still needs to consider and prioritize the underlying hardware when designing a lock.

Avoid overloading APIs. As discussed in §5.5, our initial implementation of the SCL and AMP algorithms only used reordering mechanisms to enforce both customized policy and the NUMA-grouping algorithm. However, the profiling results show that complex policy in the reordering API does not work at a high thread count. The root cause of this issue is that the node reordering mechanism in ShflLock and CNA cannot strictly prevent certain nodes from acquiring a lock once they join the queue. Specifically, the reordering mechanism makes the scheduling decision as soon as it encounters the first suitable candidate, without considering whether a better candidate exists in the entire waiting queue. With the current lock implementation, we believe the best way to address this issue is to avoid overloading APIs. Specifically, a lock developer should not specify too complicated policies in one API and, if possible, divide the policies into small pieces and enforce each one of them with the suitable APIs. For example, in our AMP implementation, we enforced the NUMA grouping policy through reordering and the fast core scheduling policy when cores enter the wait queue.

6 Discussion

6.1 Generality of SynCord

SynCord's current implementation focuses on non-blocking locks, but the fundamental concept can be applied to other synchronization primitives, such as blocking locks (mutex) [55], RCU [50], seqlocks [27], and wait events [68]. SynCord can similarly modularize key decisions and behaviors of these synchronization primitives and expose them as APIs. For example, SynCord may expose the condition to wake up or park a thread as APIs for a blocking lock. Moreover, with *lock bypass* APIs, which grant privileged users complete control of the kernel lock, the implementation of kernel locks can even be moved to user space.

6.2 Support for multi-tenancy

The current SynCord prototype targets an environment where one user or a set of users trust each other to share the machine. In this scenario, SynCord can resolve patch conflicts (i.e., a lock instance is patched by multiple patches) by allowing a privileged user to provide a final patch (§3.3). However, this approach no longer works in a multi-tenancy cloud environment.

To extend SynCord to a multi-tenancy environment, we plan to apply the widely used *cgroup* and *namespace* concepts to kernel synchronization primitives. For example, a synchronization cgroup controls the set of kernel synchronization primitives that an application can change. A synchronization namespace virtualizes the underlying synchronization primitives. For example, for a shared kernel lock, a corresponding virtual lock is created in every synchronization namespace. The kernel implements an arbitration mechanism, such as

time-sharing, to decide which virtual lock can hold the physical lock. Hence, the synchronization namespace enables applications to modify synchronization primitives while still enforcing performance isolation. With this change, we can drop privilege for SynCord to each namespace instead of limiting it to system administrators.

6.3 Easier programming of lock policy

In the current SynCord prototype, a user needs to provide the entire code for each policy. For example, both AMP (§5.2) and SCL (§5.3) policies include NUMA grouping code to achieve better scalability. We can extend SynCord to compose multiple policies into one (*i.e.*, merge NUMA-grouping and AMP to get NUMA-aware AMP) unless there is a duplicate use of the API between policies.

SynCord currently only supports C to write customized lock code, but it can be further extended to support more languages. With several toolchains [24, 30, 64] that allow writing eBPF programs in languages other than C, SynCord can support more expressive and memory-safe languages such as Python or Rust with better libraries and ecosystems.

6.4 Patching time

With our environment, the patching time is typically 10-40ms, and at a maximum of 5 seconds in an extreme case: patching every spinlock in the kernel. Livepatch applies the patch by checking each thread's stack whether the thread has invoked any patched functions. If so, Livepatch waits until all threads exit the patched function. One of the reasons for long patching time derives from a few tasks blocking the completion of a patching operation. The five-second patching time looks unreasonably long to us and further reducing the patching time is possible by sending a fake signal to the remaining blocking tasks.

7 Related Work

In section §2.2, we covered several works that customize kernel from user space. While SynCord is the first work to expose the concurrency control to user space, the need for different locking designs depending on hardware or software requirements has been also discussed in previous works.

Dice and Kogan [18] presented the `BRAVO` lock which can dynamically switch between a centralized reader-writer lock and a lock using distributed reader indicators. When the `BRAVO` algorithm detects a *read-biased* workload pattern, it improves scalability between readers by using distributed reader indicators, but at a cost of a potential slow down when released from the read-biased mode. This clear trade-off shows that the logic to turn on the read-biased mode is critical to performance, but `BRAVO` relies on heuristic parameters because it was impossible to change lock algorithms on the fly. Once ported to SynCord, when to turn on the read-biased mode can be an open problem for users.

Recently, Chehab *et al.* [17] proposed CLoF, which generates hundreds of possible combinations of spinlocks to create NUMA-aware hierarchy locks. CLoF first generates a set of locks and then selects the best performing lock for the target environment. The work emphasizes the need for different lock designs depending on the underlying hardware, which strengthens our motivation. Constructing and finding the best performing lock is an orthogonal topic to our work.

Lock profiling. Synchronization primitives play a significant role in application scalability, thus sophisticated lock profiling tools can help users understand the performance bottleneck. In addition to the `perf` [1] and `lockstat` [73] introduced in §5.5, there are several more works to improve lock profiling.

SyncPerf [3] hooks `pthread` related functions and provide a synchronization analysis tool with low overhead. Tallent *et al.* [67] used a sampling approach to quantify lock contention and SyncProf [71] collects profiling data through repetitive execution of an application to find the source of contention. Although these studies contributed to a better understanding of lock contention, all three works profile locks used in the user space, not the kernel locks.

LockDoc [43] traced the usage of kernel locks and automatically generates documentation describing the order in which each lock should be used. The work mainly focused on inferring locking rules instead of the performance aspect of each lock. wPerf [72] analyzes waiting events to find the source of a performance bottleneck, but does not provide lock-specific statistics.

8 Conclusion

Kernel synchronization primitives greatly impact application performance and scalability. However, the current kernel design prevents application developers from controlling the kernel synchronizations. This paper proposes *application-informed kernel synchronization primitives* which allow users to customize the kernel locks on the fly. To showcase the idea, we implemented SynCord, a framework for user-defined custom lock code. SynCord allows a privileged user to deploy custom code into the kernel lock safely and efficiently without recompiling or rebooting the kernel. We show that applications can leverage SynCord to achieve significant performance gains by developing hardware- or workload-specific lock algorithms. Furthermore, SynCord enables users to perform custom, fine-granularity lock profiling, which can greatly simplify the performance analysis of lock algorithms.

9 Acknowledgment

We sincerely thank our shepherd Adam Belay and the anonymous reviewers for their insightful feedback. This work is supported by Huawei.

References

[1] perf: Linux profiling with performance counters , 2014. https://perf.wiki.kernel.org/index.php/Main_page.

[2] J. Ahn, C. Kim, J. Han, Y. ri Choi, and J. Huh. Dynamic virtual machine scheduling in clouds for architectural shared resources. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA, 2012. USENIX Association.

[3] M. M. u. Alam, T. Liu, G. Zeng, and A. Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 298–313, 2017.

[4] Apple. Small chip. Giant leap., 2020. https://www.apple.com/mac/m1/.

[5] ARM. Processing Architecture for Power Efficiency and Performance, 2021. https://www.arm.com/why-arm/technologies/big-little.

[6] A. Bijlani and U. Ramachandran. Extension Framework for File Systems in User space. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 121–134, Renton, WA, July 2019.

[7] A. Blanchard. will-it-scale, 2013. https://github.com/antonblanchard/will-it-scale.

[8] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Vancouver, Canada, Oct. 2010.

[9] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[10] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 973–990, Virtual, Nov. 2020.

[11] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware Reader-writer Locks. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 157–166, Shenzhen, China, Feb. 2013.

[12] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, San Francisco, CA, Feb. 2015.

[13] J. Corbet. MCS locks and qspinlocks, 2014. https://lwn.net/Articles/590243/.

[14] J. Corbet. User-space page fault handling, 2015. https://lwn.net/Articles/636226/.

[15] J. Corbet. Controlling the CPU scheduler with BPF, 2021. https://lwn.net/Articles/873244/.

[16] I. Cutress. Intel Alder Lake: Confirmed x86 Hybrid with Golden Cove and Gracemont for 2021, 2020. https://www.anandtech.com/show/15979/intel-alder-lake-confirmed-x86-hybrid-with-golden-cove-and-gracemont-for-2021.

[17] R. L. de Lima Chehab, A. Paolillo, D. Behrens, M. Fu, H. Härtig, and H. Chen. Clof: A compositional lock framework for multi-level numa systems. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[18] D. Dice and A. Kogan. BRAVO: Biased Locking for Reader-Writer Locks. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 315–328, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8.

[19] D. Dice and A. Kogan. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 12:1–12:15, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6281-8.

[20] D. Dice and A. Kogan. Hemlock: Compact and scalable mutual exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 173–183, 2021.

[21] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 247–256, New Orleans, LA, Feb. 2012.

[22] 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 (SOSP)*, Copper Mountain, CO, Dec. 1995.

[23] M. Fleming. A thorough introduction to eBPF, 2017. https://lwn.net/Articles/740157/.

[24] foniod. RedBPF, 2021. https://github.com/foniod/redbpf.

[25] S. Ghemawat and J. Dean. LevelDB, 2019. URL https://github.com/google/leveldb.

[26] H. Guiroux, R. Lachaize, and V. Quéma. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 649–662, Denver, CO, June 2016.

[27] G. Haskins. seqlock: serialize against writers, 2008. https://lwn.net/Articles/296209/.

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

[29] Intel. Introduction to the Storage Performance Development Kit (SPDK), 2016. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-the-storage-performance-development-kit-spdk.html.

[30] iovisor. BPF Compiler Collection (BCC), 2021. https://github.com/iovisor/bcc.

[31] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099, 1998. URL https://arxiv.org/abs/cs/9809099.

[32] S. Jennings. Kernel live patching, 2014. https://lwn.net/Articles/619390/.

[33] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[34] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[35] S. Kashyap, C. Min, and T. Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[36] S. Kashyap, C. Min, and T. Kim. Scaling Guest OS Critical Sections with eCS. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[37] S. Kashyap, I. Calciu, X. Cheng, C. Min, and T. Kim. Scalable and Practical Locking With Shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[38] S. Kim, H. Kim, J. Lee, and J. Jeong. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *15th USENIX Conference on File and Storage Technologies (FAST)*, pages 345–358, Santa Clara, CA, Feb. 2017.

[39] A. Kogan. [PATCH 0/3] Add NUMA-awareness to qspinlock, 2019. URL https://lkml.org/lkml/2019/1/30/1191.

[40] Linux. Shadow Variables, 2018. https://www.kernel.org/doc/Documentation/livepatch/shadow-vars.txt.

[41] N. Liu, J. Gu, D. Tang, K. Li, B. Zang, and H. Chen. Asymmetry-aware Scalable Locking. *CoRR*, abs/2108.03355, 2021. URL https://arxiv.org/abs/2108.03355.

[42] R. Liu, H. Zhang, and H. Chen. Scalable Read-mostly Synchronization Using Passive Reader-writer Locks. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, pages 219–230, Philadelphia, PA, June 2014.

[43] A. Lochmann, H. Schirmeier, H. Borghorst, and O. Spinczyk. Lockdoc: Trace-based analysis of locking in the linux kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, 2019.

[44] W. Long. qrwlock: Introducing a queue read/write lock implementation, 2014. URL https://lwn.net/Articles/579729/.

[45] W. Long. qspinlock: Introducing a 4-byte queue spinlock, 2014. https://lwn.net/Articles/582897/.

[46] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.*, 33(4):13:1–13:62, Jan. 2016.

[47] Madhavapeddy, Anil and Scott, David J. Unikernels: The Rise of the Virtual Library Operating System. *Commun. ACM*, page 61–69, Jan. 2014. ISSN 0001-0782.

[48] Y. Mao, R. Morris, and F. M. Kaashoek. Optimizing MapReduce for Multicore Architectures. In *MIT CSAIL, Technical Report*, 2010.

[49] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, M. Ryan, E. Rubow, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[50] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-Copy Update. In *Ottawa Linux Symposium*, OLS, 2002.

[51] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

[52] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[53] C. Min, W.-H. Kang, M. Kumar, S. Kashyap, S. Maass, H. Jo, and T. Kim. SOLROS: A Data-Centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, Porto, Portugal, Apr. 2018.

[54] I. Molnar. Linux rwsem, 2006. http://www.makelinux.net/ldd3/chp-5-sect-3.

[55] I. Molnar and D. Bueso. Generic Mutex Subsystem, 2016. https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[56] ORACLE. Ksplice, 2018. https://ksplice.oracle.com.

[57] S. Park, I. Calciu, T. Kim, and S. Kashyap. Contextual Concurrency Control. In *18th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XVIII)*, Virtual, May 2021.

[58] Y. Patel, L. Yang, L. Arulraj, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Avoiding Scheduler Subversion Using Scheduler-Cooperative Locks. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, Virtual, Apr. 2020.

[59] S. Peter, J. Li, I. Zhang, D. R. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[60] J. Poimboeuf. kpatch: dynamic kernel patching, 2014. https://lwn.net/Articles/597123/.

[61] T. L. F. Projects. DPDK, 2021. https://www.dpdk.org/.

[62] Z. Radovic and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1871-0.

[63] RedHat. Achieving high-performance, low-latency networking with xdp, 2021. https://developers.redhat.com/blog/2018/12/06/achieving-high-performance-low-latency-networking-with-xdp-part-1/.

[64] rust bpf. Rust-bcc, 2021. https://github.com/rust-bpf/rust-bcc.

[65] M. Rybczyńska. Bounded loops in bpf for the 5.3 kernel, 2019. https://lwn.net/Articles/794934/.

[66] J. Slaby. kGraft, 2014. https://lwn.net/Articles/603185/.

[67] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 269–280, jan 2010.

[68] L. Torvalds. Linux Wait Queues, 2005. http://www.tldp.org/LDP/tlk/kernel/kernel.html.

[69] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, pages 11–11, Portland, OR, June 2011.

[70] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[71] T. Yu and M. Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 389–400, 2016.

[72] F. Zhou, Y. Gan, S. Ma, and Y. Wang. wPerf: Generic Off-CPU analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, Carlsbad, CA, Oct. 2018.

[73] P. Zijlstra. lockstat: documentation, 2003. https://lwn.net/Articles/252835/.

BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems

Alexander Van't Hof
Columbia University

Jason Nieh
Columbia University

Abstract

Containers are widely deployed to package, isolate, and multiplex applications on shared computing infrastructure, but rely on the operating system to enforce their security guarantees. This poses a significant security risk as large operating system codebases contain many vulnerabilities. We have created BlackBox, a new container architecture that provides fine-grain protection of application data confidentiality and integrity without trusting the operating system. BlackBox introduces a container security monitor, a small trusted computing base that creates protected physical address spaces (PPASes) for each container such that there is no direct information flow from container to operating system or other container PPASes. Indirect information flow can only happen through the monitor, which only copies data between container PPASes and the operating system as system call arguments, encrypting data as needed to protect interprocess communication through the operating system. Containerized applications do not need to be modified, can still make use of operating system services via system calls, yet their CPU and memory state are isolated and protected from other containers and the operating system. We have implemented BlackBox by leveraging Arm hardware virtualization support, using nested paging to enforce PPASes. The trusted computing base is a few thousand lines of code, many orders of magnitude less than Linux, yet supports widely-used Linux containers with only modest modifications to the Linux kernel. We show that BlackBox provides superior security guarantees over traditional hypervisor and container architectures with only modest performance overhead on real application workloads.

1 Introduction

Containers are widely deployed to package, isolate, and multiplex applications on shared computing infrastructure. They are increasingly used in lieu of hypervisor-based virtual machines (VMs) because of their faster startup time, lower resource footprint, and better I/O performance [6, 15, 26, 47]. Popular container mechanisms such as Linux containers rely on a commodity operating system (OS) to enforce their security guarantees. However, commodity OSes such as Linux are huge, complex, and imperfect pieces of software. Attackers that successfully exploit OS vulnerabilities may gain unfettered access to container data, compromising the confidentiality and integrity of containers—an undesirable outcome for both computing service providers and their users.

Modern systems incorporate hardware security mechanisms to protect applications from an untrusted OS, such as Intel Software Guard Extensions (SGX) [30] and Arm TrustZone [2], but they require rewriting applications and may impose high overhead to use OS services. Some approaches have built on these mechanisms to protect unmodified applications [7] or containers [3]. Unfortunately, they suffer from high overhead, incomplete and limited functionality, and massively increase the trusted computing base (TCB) through a library OS or runtime system, potentially trading one large vulnerable TCB for another.

As an alternative, hypervisors have been augmented with additional mechanisms to protect applications from an untrusted OS [11, 12, 27, 35, 67]. This incurs the performance overhead of hypervisor-based virtualization, which containers were designed to avoid. The TCB of these systems is significant, in some cases including an additional commodity host OS, providing additional vulnerabilities to exploit to compromise applications. Theoretically, these approaches could be applied to microhypervisors [10, 61] with smaller TCBs. Unfortunately, microhypervisors still inherit the complexity of hypervisor-based virtualization, including virtualizing and managing hardware resources. The reduction in TCB is achieved through a much reduced feature set and limited hardware support, making their deployment difficult in practice.

To address this problem, we have created BlackBox, a new container architecture that provides fine-grain protection of application data confidentiality and integrity without the need to trust the OS. BlackBox introduces a *container security monitor (CSM)*, a new mechanism that leverages existing hardware features to enforce container security guarantees in a small

trusted computing base (TCB) in lieu of the OS. The monitor creates protected physical address spaces (PPASes) for each container to enforce physical memory access controls, but provides no virtualization of hardware resources. Physical memory mapped to a container's PPAS is not accessible outside the PPAS, providing physical memory isolation among containers and the OS. Since container private data in physical memory only resides on pages in its own PPAS, its confidentiality and integrity is protected from the OS and other containers.

The CSM repurposes existing hardware virtualization support to run at a higher privilege level and create PPASes, but is itself not a hypervisor and does not virtualize hardware. Instead, the OS continues to access devices directly and remains responsible for allocating resources. This enables the CSM to be minimalistic and simple while remaining performant. By supporting containers directly without virtualization, no additional guest OS or complex runtime needs to run within the secured execution environment, minimizing the TCB within the container itself.

Applications running in BlackBox containers do not need to be modified and can make use of OS services via system calls, with the added benefit of their data being protected from the OS. The monitor interposes on all transitions between containers and the OS, clearing container private data in CPU registers and switching PPASes as needed. The only time in which any container data in memory is made available to the OS is as system call arguments, which only the monitor itself can provide by copying the arguments between container PPASes and the OS. The monitor is aware of system call semantics and encrypts system call arguments as needed before passing them to the OS, such as for interprocess communication between processes, protecting container private data in system call arguments from the OS. Given the growing use of end-to-end encryption for I/O security [55], in part due to the Snowden leaks [36], the monitor relies on applications to encrypt their own I/O data to simplify its design. Once a system call completes and before allowing a process to return to its container, the monitor checks the CPU state to authenticate the process before switching the CPU back to using the container's PPAS.

In addition to ensuring a container's CPU and memory state is not accessible outside the container, BlackBox protects against malicious code running inside containers. Only trusted binaries, which are signed and encrypted, can run in BlackBox containers. The monitor is required to decrypt the binaries, so they can only run within BlackBox containers with monitor supervision. The monitor authenticates the binaries before they can run, so untrusted binaries cannot run in BlackBox containers. It also guards against memory-related Iago attacks, attacks that maliciously manipulate virtual and physical memory mappings, that could induce arbitrary code execution in a process in a container by preventing virtual or physical memory allocations that could overwrite a process's stack.

We have implemented BlackBox on Arm hardware, given Arm's growing use in personal computers and cloud computing infrastructure along with its dominance on mobile and embedded systems. We leverage Arm hardware virtualization support by repurposing Arm's EL2 privilege level and nested paging, originally designed for running hypervisors, to enforce separation of PPASes. Unlike x86 root operation for running hypervisors, Arm EL2 has its own hardware system state. This minimizes the cost of trapping to the monitor running in EL2 when calling and returning from system calls because system state does not have to be saved and restored on each trap. We show that BlackBox can support widely-used Linux containers with only modest modifications to the Linux kernel, and inherits support for a broad range of Arm hardware from the OS. The implementation has a TCB of less than 5K lines of code plus a verified crypto library, orders of magnitude less than commodity OSes and hypervisors. With such a reduced size, the CSM is significantly easier for developers to maintain and ensure the correctness of than even just the core virtualization functionality of a hypervisor. We show that BlackBox can provide finer granularity and stronger security guarantees than traditional hypervisor and container architectures with only modest performance overhead for real application workloads.

2 Threat Model and Assumptions

Our threat model is primarily concerned with OS vulnerabilities that may be exploited to compromise the confidentiality or integrity of a container's private data. Attacks in scope include compromising the OS or any other software to read or modify private container memory or register state, including by controlling DMA-capable devices, or via memory remapping and aliasing attacks. We assume a container does not voluntarily reveal its own private data whether on purpose or by accident, but attacks from other compromised containers, including confidentiality and integrity attacks, are in scope. Availability attacks by a compromised OS are out of scope. Physical or side-channel attacks [5, 32, 43, 52, 68, 69] are beyond the scope of the paper. Opportunities for side-channel attacks are greater in BlackBox than in systems that isolate at a lower level, e.g. VMs. The trust boundary of BlackBox is that of the OS's system call API, enabling adversaries to see some details of OS interactions such as sizes and offsets.

We assume secure key storage is available, such as provided by a Trusted Platform Module (TPM) [31]. We assume the hardware is bug-free and the system is initially benign, allowing signatures and keys to be securely stored before the system is compromised. We assume containers use end-to-end encrypted channels to protect their I/O data [21, 37, 55]. We assume the CSM does not have any vulnerabilities and can thus be trusted; formally verifying its codebase is future work. We assume it is computationally infeasible to perform brute-force attacks on any encrypted container data, and any encrypted communication protocols are assumed to be designed to defend against replay attacks.

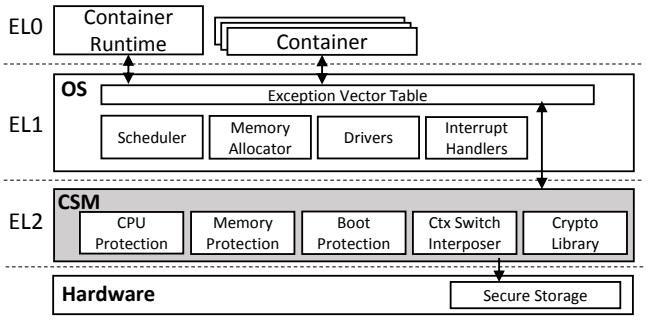


Figure 1: BlackBox Architecture

3 Design

BlackBox enclaves traditional Linux containers to protect the confidentiality and integrity of container data. We refer to a container as being enclaved if BlackBox protects it from the OS. From an application's perspective, using enclaved containers is little different from using traditional containers. Applications do not need to be modified to use enclaved containers and can make use of OS services via system calls. Container management solutions [48, 49] such as Docker [20] can be used to manage enclaved containers. BlackBox is designed to support commodity OSes, though minor OS modifications are needed to use its enclave mechanism, in much the same way that OS modifications are typically required to take advantage of new hardware features. However, BlackBox does not trust the OS and a compromised OS running enclaved containers cannot violate their data confidentiality and integrity.

BlackBox introduces a container security monitor (CSM), as depicted in Figure 1, which serves as its TCB. The CSM's only purpose is to protect the confidentiality and integrity of container data in use. It achieves this by performing two main functions, access control and validating OS operations. Its narrow purpose and functionality makes it possible to keep the CSM small and simple, avoiding the complexity of many other trusted system software components. For example, unlike a hypervisor, the CSM does not virtualize or manage hardware resources. It does not maintain virtual hardware such as virtual CPUs or devices, avoiding the need to emulate CPU instructions, interrupts, or devices. Instead, interrupts are delivered directly to the OS and devices are directly managed by the OS's existing drivers. It also does not do CPU scheduling or memory allocation, making no availability guarantees. The CSM can be kept small because it presumes the OS is CSM-aware and relies on the OS for complex functionality such as bootstrapping, CPU scheduling, memory management, file systems, and interrupt and device management.

To enclave containers, the CSM introduces the notion of a *protected physical address space (PPAS)*, an isolated set of physical memory pages accessible only to the assigned owner of the PPAS and the CSM. Each page of physical memory is mapped to at most one PPAS. The CSM uses this mechanism to provide memory access control by assigning a separate PPAS to each enclaved container, thereby isolating the physical memory of each container from the OS and any other container. The OS determines what memory is allocated to each PPAS, but cannot access the memory contents of a PPAS. Similarly, a container cannot access a PPAS that it does not own. Memory not assigned to a PPAS, or the CSM, is assigned to and accessible to the OS. The CSM itself can access any memory, including memory assigned to a PPAS. Within a PPAS, addresses for accessing memory are the same as the physical addresses on the machine; physical memory cannot be remapped to a different address in a PPAS. For example, if page number 5 of physical memory is assigned to a PPAS, it will be accessed as page number 5 from within the PPAS. Container private data in memory only resides on pages mapped to its own PPAS, therefore its confidentiality and integrity is protected from the OS and other containers. Section 4 describes how BlackBox uses nested page tables to enforce PPASes.

The CSM interposes on all transitions between containers and the OS, namely system calls, interrupts, and exceptions, so that it can ensure that processes and threads, which we collectively refer to as tasks, can only access the PPAS of the container to which they belong when executing within context of the container. The CSM ensures that when a task traps to the OS and switches to running OS kernel code, the task no longer has access to the container's PPAS. Otherwise, the OS could cause the task to access the container's private data, compromising its confidentiality or integrity. The CSM maintains an *enclaved task array*, an array with information for all tasks running in enclaved containers. When entering the OS, the CSM checks if the calling task is in an enclaved container, in which case it saves to the enclaved task array the CPU registers and the cause of the trap, switches out of the container's PPAS, and clears any CPU registers not needed by the OS. When exiting the OS, the CSM checks the enclaved task array if the running task belongs to an enclaved container, in which case it validates the current CPU context, namely the stack pointer and page table base register, match what was saved in the enclaved task array for the respective task. If they match, the CSM switches to the respective container's PPAS so the task can access its enclaved CPU and memory state. As a result, container private data in CPU registers or memory is not accessible to the OS.

To support OS functionality that traditionally requires access to a task's CPU state and memory, the CSM provides an application binary interface (ABI) for the OS to request services from the CSM. The CSM ABI is shown in Table 1. For example, `create_enclave` and `destroy_enclave` are called by the OS in response to requests from a container runtime, such as `runC` [29], to enclave and unenclave containers, respectively. For CSM calls that require dynamically allocated memory, the OS must allocate and pass in the physical address of a large enough region of contiguous memory to perform the respective operation. Otherwise, the call will fail and return the amount of memory required so that the OS can

CSM Call	Description
create_enclave	Create new enclave for a container.
destroy_enclave	Destroy enclave of a container.
protect_vectors	Validate OS exception vectors.
alloc_iopgtable	Allocate I/O device page table.
free_iopgtable	Free I/O device page table.
set_iopt	Update entry in I/O device page table.
get_ioaddr	Get physical address for I/O virtual address.
enter_os	Context switch CPU to OS.
exit_os	Context switch CPU from OS.
set_vma	Update virtual memory areas of a process/thread.
set_pt	Update page table entry of a process/thread.
copy_page	Copy contents of a page to a container.
task_clone	Run new process/thread in a container.
task_exec	Run in new address space in a container.
task_exit	Exit a process or thread in a container.
futex_read	Read the value of a futex in a container.

Table 1: BlackBox Container Security Monitor ABI

make the call again with the required allocation. For example, create_enclave requires the OS to allocate memory to be used for metadata for the enclaved container. Upon success, the allocated memory is assigned to the CSM and no longer accessible to the OS until destroy_enclave is called, at which point the memory is assigned back to the OS again.

3.1 System Boot and Initialization

BlackBox boots the CSM by relying on Unified Extensible Firmware Interface (UEFI) firmware and its signing infrastructure with a hardware root of trust. The CSM and OS kernel are linked as a single binary which is cryptographically signed, typically by a cloud provider running BlackBox containers; this is similar to how OS binaries are signed by vendors like Red Hat or Microsoft. The binary is first verified using keys already stored in secure storage, ensuring that only the signed binary can be loaded. To keep the CSM as simple as possible, BlackBox does not implement bootstrapping within the CSM itself, which can require thousands of lines of code to support many systems. Instead, it relies on the OS's bootstrapping code to install the CSM securely at boot time since the OS is initially benign. By relying on commodity OSes such as Linux that already boot on a wide range of systems, this makes it easier for the CSM to support many systems without the burden of manually maintaining and porting its own bootstrapping code for many systems.

At boot time, the OS initially has full control of the system to initialize hardware, and installs the CSM. CSM installation occurs before local storage, network and serial input services are available, so remote attackers cannot compromise its installation. Once installed, the CSM runs at a higher privilege level than the OS and subsequently enables PPASes as needed. A small amount of physical memory is statically assigned to the CSM, and the rest is assigned to the OS. Any attempt to access the CSM's memory except by the CSM itself will trap to the CSM and be rejected. Although the OS's memory is separate from the CSM's, the CSM can access the OS's memory and can restrict its from modifying its own memory if needed.

The CSM expects the hardware to include an IOMMU to protect against DMA attacks by devices managed by the OS [62]. The CSM retains control of the IOMMU and requires the OS to make CSM calls to update IOMMU page table mappings, which are typically configured by the OS during boot. This ensures that I/O devices can only access memory mapped into the IOMMU page tables managed by the CSM. The OS calls alloc_iopgtable during boot to allocate an IOMMU translation unit and its associated page table for a device, and set_iopt and to assign physical memory to the device to use for DMA. The CSM ensures that the OS can only assign its own physical memory to the IOMMU page tables, ensuring that DMA attacks cannot be used to compromise CSM or container memory.

3.2 Enclaved Container Initialization

To securely initialize an enclaved container, an image that is to be used for such a container must first be processed into a BlackBox container image, using a process similar to how Amazon enclaves are created using Docker images [1]. BlackBox provides a command line tool build_bb_image, which can be used by a cloud customer, that takes a Docker image, finds all executable binary files contained within the image, and encrypts the sections containing the code and data used by the code using the public key paired with a trusted private key stored in the secure storage of the host and accessible only by the CSM. These encrypted sections are then hashed and their hash values recorded along with the binaries they belong to. These values are then signed with the private key of the container image's creator whose paired public key is accessible in the secure storage of the host to ensure authenticity and bundled with the container image for later reference during process creation, as described in Section 3.3. This ensures the binaries cannot be modified without being detected, or run unless decrypted by the CSM. Other than additional hashes and using encrypted binaries, the BlackBox container image contains nothing different from a traditional Docker image.

To start a container using a BlackBox container image, the container runtime is modified to execute a simple shim process in place of the container's specified init process. The container runtime passes the shim the path of the init process used by the container along with any arguments and its environment. The shim is also given the signed binary hash information bundled with the container image. The shim process runs a tiny statically linked program that initiates a request to the OS to call the create_enclave CSM call before executing the original init process, passing the signed hash information to the CSM as part of the call. Other than the shim process, which exits upon executing the init process, there is no additional code that runs in a BlackBox container

beyond vanilla Linux containers. There are no additional libraries and no need for a library OS, avoiding the risks of bloating the TCB of the container itself.

`create_enclave` creates a new enclave using the BlackBox container image and returns with the calling process running in the enclaved container, the return value of the call being the new enclave's identifier. `create_enclave` performs the following steps. First, it creates a new PPAS for the container. Second, it freezes the userspace memory of the calling process so it, and its associated page tables, cannot be directly changed by the OS, then moves all of its pages of physical memory into the container's PPAS so that they are no longer accessible by the OS. Finally, it checks the contents of the loaded shim binary in memory against a known hash to validate the calling process is the expected shim process.

After returning from `create_enclave`, the shim executes the container's init process from within the container. Since the container's init process obtains its executable from the BlackBox container image whose code and data are encrypted, the OS may load it, but cannot actually execute it without the CSM using its private key to decrypt it. Further details on `exec` with encrypted binaries are described in Section 3.6. In this way, the OS is incapable of running a BlackBox container image without the CSM. Therefore, if it is running, the CSM must be involved and protecting it. Because the CSM itself is securely booted and enclave code is encrypted and only runnable by the CSM, an unbroken chain of trust is established enabling remote attestation similar to that of other security systems, such as Samsung Knox [56].

The container runtime calls `destroy_enclave` to remove the enclave of a container, which terminates all running processes and threads within the container to ensure that any container CPU state and memory is cleared and no longer accessible to the OS or any other container before removing the enclave. The container is effectively returned to the same state it was in before `create_enclave` was called.

3.3 Enclaved Task Execution

BlackBox supports the full lifecycle of tasks executing in enclaved containers, including their dynamic creation and termination via standard system calls such as `fork`, `clone`, `exec`, and `exit`. This includes tracking which tasks are allowed to execute in which containers. This is achieved by requiring the OS to call a set of CSM calls, `task_clone` on task creation via `fork` and `clone`, `task_exec` when loading a new address space via `exec`, and `task_exit` when a task exits via `exit`. These calls request the CSM to perform various functions related to task execution that the OS is not able to do because it does not have access to task CPU state and memory. If the OS does not make the respective CSM call, the created task and executed binary will simply not run in its enclave and therefore will not have access to its data. These calls update the enclaved task array, the index of which is used as the enclaved task identifier. Each entry in the array includes the enclave identifier of the container in which the task executes, as well as the address of the page table used by the task as discussed earlier.

When a task running in an enclaved container creates a child task via a system call, the OS calls `task_clone` with the enclaved task identifier of the calling task and a flag indicating whether the new task will share the same address space as the caller, as when creating a thread, or have its own copy of the address space of the caller, as when creating a process. In the latter case, new page tables will be allocated for the child task and the CSM will ensure that they match those of the caller's and cannot be directly modified by the OS. The CSM will also confirm that the calling task issued the task creation system call. If all checks pass, the CSM will create a new entry in the enclaved task array with the same enclave identifier as the calling process, and return the array index of the new entry as the identifier for the task. The entry will also contain the address of the task's page table, which will be the same as the caller's entry if it shares the same address space as the caller.

When the OS runs the child and the task returns from the OS, the OS provides the CSM with the enclaved task's identifier. The CSM then looks up the task in its enclaved task array using this identifier and confirms that the address of the page table stored in the entry matches the address stored in the page table base register of the CPU. If the checks pass, it will then restore CPU state and switch the CPU to the container's PPAS, thereby allowing the task to resume execution in the container. If the OS does not call `task_clone`, then upon exiting the OS, the task's PPAS would not be installed and it would fail to run.

On `exec`, the calling task will replace its existing address space with a new one. The OS calls `task_exec`, which, like `task_clone` for `fork`, creates a new enclaved task entry with a new address space. The difference is that the new address space is validated by ensuring that the new process' stack is set up as expected and the executable binary is signed and in the BlackBox container image, as described in Section 3.6. After creating the new enclaved task entry, the original address space is disassociated from the container, scrubbing any memory pages to be returned to the OS and removing them from the container's PPAS.

On `exit`, the OS will call `task_exit` so the CSM can remove the enclaved task entry from the enclaved task array. If an address space has no more tasks in the container, the CSM disassociates it in a similar manner to the `exec` case.

3.4 Memory

BlackBox prevents the OS from directly accessing a container's memory, but relies on the OS for memory management, including allocating memory to tasks in the container. This avoids introducing complex memory management code into BlackBox, keeping it small and simple, but means that BlackBox also needs to protect against memory-based Iago

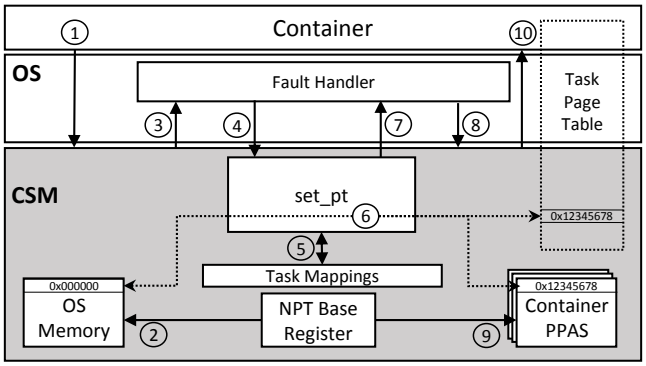


Figure 2: BlackBox Page Fault Workflow

attacks [9] by the untrusted OS through manipulation of system call return values. For example, if a process calls `mmap`, it expects to receive an address mapping that does not overlap with any of its existing mappings. If the OS were to return a value overlapping the process's stack, it could manipulate the process into overwriting a return address on its stack through a subsequent `read` with an attacker controlled address, opening the door for return-oriented-programming [53] and return-into-libc [58] attacks. Furthermore, the OS may return an innocuous looking non-overlapping virtual address from `mmap`, but still maliciously map the returned address to the physical page the stack is on.

To rely on the OS for memory management while preventing memory-based Iago attacks, BlackBox protects the container's memory at the application level by preventing the OS from directly updating per process page tables. It instead requires the OS to make requests to the CSM to update process page tables, allowing the CSM to reject updates if the OS behaves incorrectly. Figure 2 depicts how a container's page table is updated during a page fault. When a process in a container faults on a page, an exception causes control to transfer to the OS by way of the CSM (steps 1-3). The OS then allocates a page for the process, but instead of updating the process page table directly, it performs a `set_pt` CSM call (step 4). Upon receiving the `set_pt` call, the CSM verifies if the allocation is acceptable (step 5). To do so, the CSM maintains a list of valid mappings for each process. This list is maintained by interposing on system calls that adjust memory mappings. In Linux these calls include `mmap` and `brk`. Prior to writing the page table entry, the CSM first verifies that the virtual address specified belongs to a valid mapping. If it does not, the update is rejected. Second, the CSM checks if the physical page assigned is already in the container's PPAS and therefore already in use. This can commonly occur innocuously when, e.g., two processes in a container have the same file mapped in their address spaces. However, to prevent the risk of a malicious OS coercing an enclave to overwrite existing memory via a malicious memory allocation, the CSM marks any physical page mapped more than once read only in the container's PPAS, unless it was inherited from a parent as part of process creation in which case it can be trusted. While this is effective at preventing these attacks, the downside is that writes to such memory will trap and need to be handled by BlackBox; for simplicity, BlackBox disallows writable memory-mapped file I/O as it is uncommonly used. Finally, if the virtual address is valid and not mapped to an existing physical page in a container's PPAS, the CSM unmaps the assigned physical page from the OS and maps it into the container's PPAS. The CSM then updates the page table entry on the OS's behalf (step 6). Control is then returned back to the OS (step 7). When returning control back to the process that faulted, the process's container PPAS will be switched to (steps 8-10). Section 4 describes further details about this process. The CSM also invalidates TLB entries as needed when it performs page table updates, ensuring that a malicious OS cannot violate a container's PPAS through stale TLB entries.

BlackBox provides support for copy-on-write (CoW) memory, a key optimization commonly used in OSes. The OS traditionally expects to be able to share a page in memory among multiple processes and when a write is attempted, break the CoW by copying the contents of the page to a new page assigned to the process. With BlackBox, the OS does not have the ability to copy container memory though, so the OS instead makes a `copy_page` CSM call to have the CSM perform the CoW break on its behalf. The CSM will check that the source page belongs to the container's PPAS and the destination page is in the OS's memory. If so, it will move the destination page into the container's PPAS and perform the copy.

BlackBox supports the dynamic release of memory back to the OS as tasks adjust their heap, unmap memory regions, and exit, while preserving the privacy and integrity of a container's memory. As with memory allocation, system calls that can allow for returning of an application's memory, like `munmap` and `_exit` are tracked to maintain an accurate view of a container's memory mappings. During these calls, the OS may attempt to free pages allocated to the process. In doing so, as with memory allocation, it must make use of the `set_pt` CSM call since it cannot update page tables directly. The CSM will then check if the application has made a call to release the specified memory and reject the update if it has not. If the update is valid, the CSM will perform the page table update, and if no longer needed, scrub the page and remove it from the container's PPAS.

While BlackBox ensures that container memory is not accessible to the OS, many OS interactions via system calls expect to use memory buffers that are part of an application's memory to send data to, or receive data from, the OS. BlackBox treats the use of such memory buffers in system calls as implicit directives to declassify the buffers so they can be shared with the OS. To support this declassification while ensuring that a container's PPAS is not accessible to the OS, BlackBox provides a syscall buffer for each task running in an enclaved container that is outside of the container's PPAS and accessible to the OS. When interposing

on a system call exception, the CSM replaces references to memory buffers passed in as system call arguments with those to the task's syscall buffer. For buffers that are used to send data to the OS, the data in those buffers is copied to the syscall buffer as well. When returning to the container, the references to the syscall buffer are replaced with those to the original memory buffers. For buffers that are used to receive data from the OS, the data in the syscall buffer is copied to the original memory buffers as well.

Most system calls are interposed on by a single generic wrapper function in the CSM that uses a table of system call metadata to determine which arguments must be altered. System calls with more complex arguments, like those involving `iovec` structures are interposed on with more specific wrapper functions. On Linux, this interposing and altering of arguments works for most system calls with a few notable exceptions as discussed in Section 3.5.

As part of the copying of data from the OS to an enclaved container, BlackBox also does simple checks on system call return values to ensure they fall within predefined correct ranges. This has been shown to protect against many Iago attacks [14]. However, to keep its TCB simple and small, BlackBox only guarantees the correctness of system call semantics for memory management and inter-process communication (IPC), the latter discussed in Section 3.5. As a result, BlackBox protects against Iago attacks related to memory management and IPC, but is susceptible to some other Iago attacks. Augmenting BlackBox with a user-level runtime library in an enclaved container that guarantees the correctness of system call semantics could improve Iago attack protection, but at the cost of a larger TCB and potential additional limitations on system call functionality.

3.5 Inter-process Communucation

While BlackBox declassifies data to the OS passed in as system call arguments, it protects inter-process communication (IPC) among tasks running in the same enclaved container by encrypting the data passed into IPC-related system calls. This protects applications using IPC, which is transferred through and accessible to the OS. System calls that can create IPC-related file descriptors, such as `pipe`, and Unix Domain Sockets are interposed on and their returned file descriptors (FDs) recorded in per-process arrays marking them as related to IPC. When the CSM interposes on system calls that pass data through FDs, like `write` and `sendmsg`, it checks if the given FD is one related to IPC for that process. If it is, the CSM first uses authenticated encryption with a randomly generated symmetric key created during container initialization to encrypt the data before moving it into the task's syscall buffer. A record counter, incremented on each transaction, is included as additional authenticated data to prevent the host from replaying previous transactions. Similarly, data is decrypted and authenticated when interposing on system calls like `read` and `recvmsg` before copying it to the calling process's PPAS. With this mechanism, IPC communication is transparently encrypted and protected from the OS.

As mentioned in Section 3.4, to avoid trusting the OS's memory allocations, memory pages that are used by more than one process in a container are marked read-only in the container's PPAS unless the pages are known to belong to a shared memory mapping and are inherited during process creation. Shared memory regions created by a parent process through `mmap` with `MAP_SHARED` and faulted in prior to forking can be written to by both parent and child processes since the child's address space is validated after `fork`, as discussed in Section 3.3. However, for simplicity, BlackBox does not allow for writable IPC shared memory via XSI IPC methods such as `shmget` and `shm_open`, which are no longer widely-used. Modern applications instead favor thread-based approaches for performance or shared mappings between child worker processes via `mmap` compatible with BlackBox.

Futexes are used among threads and processes to synchronize access to shared regions of memory. As part of the design of `futex`, the OS is required to read the futex value, which is in the process's address space and included in the respective container's memory. This direct access to container memory is incompatible with BlackBox's memory isolation. To support `futex`, the OS makes a `futex_read` CSM call to obtain the value of a futex for container processes, rather than try and access the memory directly. The CSM ensures that only the futex address passed to `futex` can be read, and only if a `futex` call has been made.

Signals, used to notify processes of various events, present two issues for BlackBox. First, when delivering a signal to a process, a temporary stack for the signal handler is set up in the process's memory. With enclaved containers, this memory is not accessible to the OS. To remedy this, the OS is modified to setup this stack in a region of memory outside of the container's PPAS, which is then moved to the PPAS when the signal handler is executed and returned to the OS when the signal handler returns via `rt_sigreturn`. Second, the OS has to adjust the control flow of the process to execute the signal handler instead of returning to where it was previously executing. BlackBox cannot allow the OS to adjust the control flow of an enclaved process without validating it is doing so properly. To achieve this, as part of the CSM interposing on system calls, it tracks signal handler installation via system calls such as `rt_sigaction`. Upon handling a signal, the CSM ensures that the process will be correctly returning to a registered handler.

3.6 Container File System

Files within a container can only be accessed through an OS's I/O facilities making access to a container's files inherently untrustworthy without additional protection. A userspace encrypted file system could potentially be used to provide transparent protection of file I/O, but this would likely signif-

icantly increase the container's TCB. BlackBox relies on applications to use encryption to fully protect sensitive data files within a container, and provides a simple mechanism to allow the OS to load encrypted executable binaries for execution.

As discussed in Section 3.2, container images for BlackBox are pre-processed. For example, ELF binaries, widely-used on Linux, have `.text`, `.data`, and `.rodata` sections that contain the executable code and data used by the code. These sections are combined into various segments when loaded into memory. In a BlackBox container image, the ELF headers are left unencrypted, but the `.text`, `.data`, and `.rodata` sections are encrypted then hashed, and their hash values are recorded along with the binaries. This enables BlackBox to validate the integrity and authenticity of executable binaries.

An ELF binary is executed by the OS as a result of a process calling `exec`, upon which the OS loads the binary by mapping its ELF headers into memory, reading the ELF headers to determine how to process the rest of the binary, then mapping the segments of the binary to memory. As discussed in Section 3.3, the OS is required to call `task_exec`, which passes the virtual addresses of the binary's loaded segments containing the `.text`, `.data`, and `.rodata` sections to the CSM. During this call, the CSM moves the process's pages, corresponding to the loaded binary, into the container's PPAS, validates that the hashes of the encrypted `.text`, `.data`, and `.rodata` sections match the hashes for the given binary from the BlackBox container image to confirm the authenticity and integrity of the loaded segments, then decrypts the sections in memory. The virtual to physical address mappings of these binary segments are recorded for later use. Upon returning from `task_exec`, the OS will begin running the task whose binary is now decrypted within protected container memory. If checking the hashes or decryption fails, the CSM will refuse to run the binary within an enclaved container, ensuring only trusted binaries can run within.

For dynamically linked binaries, in addition to the binary segments the OS maps during `exec`, the OS also maps the segments of the loader in the process's address space. These segments are verified in the same manner as the binary's segments. Dynamically linked binaries load and execute external libraries that BlackBox must validate are as expected and trusted. During the container image creation process, as with executable binaries, library binaries are also encrypted preventing their use without the CSM. These libraries are loaded and linked at runtime in userspace by a loader that is part of the trusted container image. To do this, the loader, running as part of a process's address space, `mmap`s library segments into memory. The CSM intercepts these mmaps by interposing on FD-related system calls, such as `open`. If an FD is created for one of the libraries within a container, as recorded during container image creation, the CSM marks that FD as associated with the given library. If this FD is then used with `mmap`, the CSM intercepts it. Based on the size of the mmap request and the protection flags used, the CSM can infer which segment the loader is mapping. If it is a segment containing one of the encrypted sections, the CSM performs the same hashing, decryption, and memory map recording as it does with executable binaries.

4 Implementation

We have implemented a BlackBox prototype by repurposing existing hardware virtualization support available on modern architectures, including a higher privilege level, usually reserved for hypervisors, and nested page tables (NPTs). NPTs, also known as Arm's Stage 2 page tables and Intel's Extended Page Tables (EPT), is a hardware-assisted virtualization technology that introduces an additional level of virtual address translation [8]. When NPTs are used by hypervisors, the guest OS in a VM manages its own page table to translate a virtual address to what the VM perceives as its physical address, known as a guest physical address, but then the hypervisor manages an NPT to translate the guest physical address to an actual physical address on the host. Hypervisors can thereby use NPTs to control what physical memory is available to each VM.

BlackBox uses hardware virtualization support to run the CSM in lieu of a hypervisor to support PPASes. The CSM runs at the higher hypervisor privilege level, so that it is strictly more privileged than the OS and is able to control NPTs. The CSM introduces an NPT for each container and the OS, such that a container's PPAS is only mapped to its own NPT, isolating the physical memory of each container from the OS and each other. The CSM switches a CPU from one PPAS to another by simply updating its NPT base register to point to the respective container's NPT. Similarly, the CSM uses NPTs to protect its own memory from the OS and containers by simply not mapping its own memory into the NPTs. The memory for the NPTs is part of the CSM's protected memory and is itself not mapped into any NPTs so that only the CSM can update the NPTs. When the CSM runs, NPTs are disabled, so it has full access to physical memory.

Specifically, BlackBox uses Arm hardware virtualization extensions (VE) [16–19]. The CSM runs in Arm's hypervisor (EL2) mode, which is strictly more privileged than user (EL0) and kernel (EL1) modes. EL2 has its own execution context defined by register and control state, and switching the execution context of EL0 and EL1 are done in software. The CSM configures Stage 2 page tables in EL2, and the System Memory Management Unit (SMMU), Arm's IOMMU. The Linux kernel runs in EL1 and has no access to EL2 registers, so it cannot compromise the CSM. CSM calls are made using Arm's `hvc` instruction from EL1.

Before and after every transition to the OS, BlackBox traps to the CSM, which in turn switches between container and OS NPTs. One might think that imposing two context switches to the CSM to swap NPTs for every one call to the OS would be prohibitively expensive, but we show in Section 5

that this can be done on Arm without much overhead. The flexibility that Arm EL2 provides of allowing software to determine how execution context is switched between hypervisor and other modes turns out to be particularly advantageous for implementing the CSM because it does not lock its implementation into using heavyweight hardware virtualization mechanisms to save and restore hypervisor execution context that are not required for the CSM.

Trapping to the CSM before and after every transition to the OS requires that the CSM interpose on all system calls, interrupts, and exceptions. Hypervisors traditionally accomplish similar functionality by trapping interrupts and exceptions to itself, then injecting virtual interrupts and exceptions to a VM. BlackBox avoids the additional complexity of virtualizing interrupts and exceptions by taking a different approach. The CSM configures hardware so system calls, interrupts, and exceptions trap to the OS and modifies the OS's exception vector table for handling these events so that `enter_os` and `exit_os` CSM calls are always made before and after the actual OS event handler. To guarantee these handlers are installed and not modified by the OS at a later time, BlackBox requires the OS to make a `protect_vectors` CSM call with the address of the text section of the vector table during system initialization, before any container may be enclaved. The CSM then prevents the OS from tampering with the modified vector table by marking its backing physical memory read only in the OS's NPT. Similarly, the vDSO region of memory is marked read only to prevent malicious tampering of the region.

Figure 3 depicts the steps involved in interposing on transitions between the containers and OS when repurposing virtualization hardware. While running in a container, an exception occurs transferring control to the protected OS exception vector table (step 1). All entry points in the exception vector table invoke the `enter_os` CSM call (step 2). During this, the CSM switches to the OS's NPT (step 3). The OS will therefore not be able to access private physical memory mapped into container NPTs. For system call exceptions, system call arguments are copied to an OS accessible syscall buffer (step 4). Control is transferred back to the OS (step 5) to perform the required exception handling. When the OS has finished handling the exception, the `exit_os` CSM call is made as part of the return path of the exception vectors when returning to userspace (step 6). For system call exceptions, OS updated arguments are copied back to the original buffer (step 7). On `exit_os`, the CSM verifies the exception return address to ensure the call is from the trusted exception vectors, which the OS cannot change, rejecting any that are not. The CSM then checks if the running task belongs to an enclaved container, in which case the CSM switches to the respective container's NPT so the task can access its PPAS memory state (step 8). Control is restored to the container by returning from `exit_os` (step 9) and back to userspace (step 10). If `exit_os` is not called, the CSM will not switch the CPU to use the container's PPAS, so its state will remain inaccessible on that CPU.

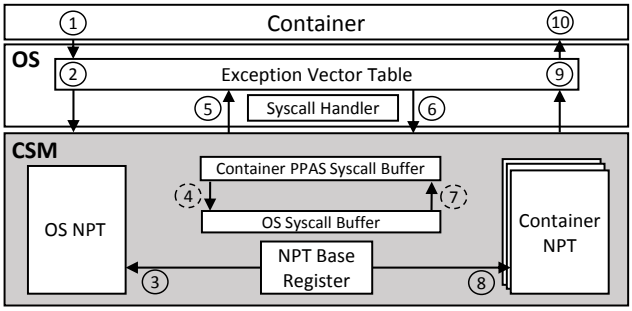


Figure 3: System Call from Enclaved Container

BlackBox protects a container's memory by using separate NPTs for the OS and each container, but still relies on the OS to perform all complex memory management functions, such as allocation and reclamation, to minimize the complexity and size of the CSM. This is straightforward because unlike hypervisors which virtualize physical memory using NPTs, the CSM merely uses NPTs for access control so that the identity mapping is used for all NPTs including the OS's NPT. The OS's view of memory is effectively the same as the actual physical memory for any physical memory mapped into the OS's NPT. Except for the CSM's physical memory, all physical memory is initially assigned to the OS and mapped to its NPT. When the OS allocates physical memory to processes in containers, the CSM can just unmap the physical memory from the OS's NPT and map it to the respective container's NPT at the same address. The CSM does not need its own complex allocation functionality. The CSM checks the OS's NPT to make sure that the OS has the right to allocate a given page of memory. For example, should the OS attempt to allocate a physical page belonging to the CSM, the CSM will reject the allocation and not update the OS's or container's NPT. The CSM also checks that any page allocation proposed by the OS for a container is not mapped into the IOMMU page tables and will therefore not be subject to DMA attacks, as discussed in Section 3.1.

Note that the OS is oblivious to the fact that its allocation decisions for process page tables, Arm's Stage 1 page tables, are also used for Stage 2 page tables. Furthermore, since Arm hardware first checks Stage 1 page tables before Stage 2 page tables, page faults due to the need to allocate physical memory to a process all appear as Stage 1 page faults, which are handled in the normal way by the OS's page fault handler. Since the CSM maps the physical memory to the respective Stage 1 and Stage 2 page table entries at the same time, there are no Stage 2 page faults for memory allocation.

As discussed in Section 3.4, BlackBox requires that process page tables cannot be directly modified by the OS. At the same time, commodity OSes like Linux perform many operations that involve walking and accessing process page tables. To minimize OS modifications required to use enclaved containers, BlackBox makes the process page tables readable but not writable by the OS by marking the corresponding entries in the OS's NPT read only. All existing OS code that walks

and reads process page tables can continue to function without modification, and only limited changes are required to the OS to use CSM calls for any updates to process page tables. A process's page tables are also mapped to its respective container's NPT, so they can be accessed by MMU hardware for virtual address translation while executing the process. BlackBox also maps tasks' syscall buffers, used for passing system call arguments to and from the OS, to their Stage 1 page tables. This allows OS functions designed to copy data to and from buffers in the calling process's address space to function correctly without modification. The tasks' syscall buffers themselves are only mapped to the OS's NPT, not the container's NPT, as they are shared directly only by the CSM and OS.

To optimize TLB usage, physically contiguous memory can be mapped to an NPT in blocks larger than the default 4 KB page size. The BlackBox implementation supports transparent 2 MB stage 2 block mappings by first fully populating the last-level stage 2 page table with 4 KB mappings, then folding all 512 entries into a single entry. BlackBox checks that all 512 entries are contiguous in physical memory and that the first entry is aligned to a 2 MB boundary. BlackBox will unfold a block mapping if one of the original 512 entries is unmapped, such that all 512 entries are no longer contiguous in physical memory. Similarly, BlackBox will unfold a block mapping if there is a need to change the attributes of one of the original 512 entries, such as marking it read only while other entries remain writable. This approach is advantageous over just supporting huge pages allocated by the OS because it improves TLB usage even when the OS does not use huge pages.

Although BlackBox is designed to work using existing hardware virtualization support, the upcoming Armv9 architecture with its inclusion of the Arm Confidential Compute Architecture (CCA) [41] offers alternative mechanisms that may be used for implementing BlackBox. CCA introduces secure execution environments called Realms. The memory and execution state of these Realms are inaccessible to existing privileged software like OSes and hypervisors guaranteeing their confidentiality and integrity from them. Realms are supported by a separate Realm World and managed by a Realm Management Monitor (RMM) running in EL2 within the Realm World giving it full access to Realm memory and CPU state as well as control over their execution. Although Realms are currently only designed to support VMs, it may be possible to use them to support enclaved containers by integrating the functionality of the CSM with the RMM and extending its ABI to encompass the CSM's ABI.

BlackBox's implementation is relatively small. The implementation is less than 10K lines of code (LOC), most of which is the 5K LOC for the implementation of Ed25519, ChaCha20, and Poly1305 from the verified HACL* crypto library [70]. Other than HACL*, BlackBox consisted of 4.9K LOC, all in C except for 0.4K LOC in Arm assembly. Table 2 shows a breakdown by feature. 0.3K LOC was for verifying the CSM was correctly booted and initialized. 1K LOC was for enclave management, including enclave creation and handling enclave metadata 0.1K LOC was for switching between enclaves and the OS. 0.2K LOC was for protecting data in CPU registers. 1K was for system call interposition, including marshaling of arguments. The table used for determining how to marshal system calls and check return values is dynamically generated as a single line of C code at compile time. 2.3K LOC was for memory protection, including NPT management of PPASes, Iago and DMA protection, and handling and validating page table update requests. BlackBox's CSM TCB implementation complexity is similar to other recently verified concurrent systems [39–41, 63], suggesting that it is small enough that it can be formally verified. Beyond the CSM itself, only 0.5K LOC were modified or added to the Linux kernel to support BlackBox.

Feature	BlackBox	Linux	KVM
Bootstrapping	0	8.5K	8.5K
Device Support	0	425K	425K
Filesystem Support	0	163K	163K
Process Management	0	110K	110K
Memory Management	0	60.7K	60.7K
CPU Scheduling	0	29.3K	29.3K
Networking	0	190K	0
Sound	0	89.3K	0
Process Security	0	64.7K	0
Device Virtualization	0	0	30.1K
CPU Virtualization	0	0	3.5K
VM Switch	0	0	1.2K
Cryptography	5K	19K	19K
Boot Verification	0.3K	0	0
Enclave Management	1K	0	0
Enclave Switch	0.1K	0	0
CPU Protection	0.2K	0	0
Syscall Interposition	1K	0	0
NPT Management	1K	0	2.8K
Memory Mapping Protection	0.5K	0	0
DMA Protection	0.8K	0	9K
Total	9.9K	1.2M	862K

Table 2: LOC for BlackBox, Linux, and KVM

Table 2 also compares the code complexity of BlackBox versus the Linux kernel and KVM hypervisor. This is a conservative comparison as the LOC for Linux and KVM only include code compiled into the actual binaries for one specific Arm server used for the evaluation in Section 5. Even with this conservative comparison, BlackBox is orders of magnitude less code, in part because its functionality is largely orthogonal to both OSes and hypervisors, which have much more complex functionality requirements.

5 Experimental Results

We quantify the performance of BlackBox compared to widely-used Linux containers, and demonstrate BlackBox's ability to protect container confidentiality and integrity. Ex-

Name	Description
Lmbench	`lmbench` v3.0-a9 [46] latency microbenchmarks.
Hackbench	`hackbench` [54] using Unix domain sockets and 100 process groups running in 500 loops.
Apache	`Apache` v2.4.46 server handling 100 concurrent requests from remote `ApacheBench` [64] v2.3 client, serving the 12 KB default Debian `index.html`.
HAProxy	`HAProxy` v1.8.19 server proxying 100 concurrent requests from remote `ApacheBench` [64] v2.3 client to remote `Apache` v2.4.29, serving the 82 KB `index.html` of the GCC 13.0.0 manual.
Kernbench	Compilation of the Linux 5.4 kernel using `allnoconfig` for Arm with GCC 8.3.0.
Memcached	`memcached` v1.6.9 using the `memtier` [51] benchmark v1.3.0 with default parameters.
MySQL	`MariaDB` v10.3.27, a `MySQL` fork, handling requests from remote `YCSB` [13] v0.17.0 client running workload A with 200 parallel transactions, recordcount=500K, and opcount=100K.
Netperf	`netperf` v2.6.0 [33] running `netserver` on the server and the client with default parameters in three modes: TCP_STREAM (receive throughput), TCP_MAERTS (send throughput), and TCP_RR (latency).
Nginx	`Nginx` v1.18.0 server handling 100 concurrent requests from remote `ApacheBench` [64] v2.3 client, serving the 12 KB default Debian `index.html`.

Table 3: Microbenchmarks and Application Workloads

periments were run using both Arm multiprocessor embedded system and server hardware with VE support, specifically (1) a Raspberry Pi 4 Model B with a 4-core Cortex-A72 64-bit 1.5 GHz Broadcom BCM2711 SoC, 8 GB RAM, a 250 GB Samsung 860 EVO SSD connected via USB3.0, and Gigabit Ethernet, running Raspberry Pi OS Buster (2020-08-20 Debian), and (2) an AMD Seattle Rev.B0 server with an 8-core Cortex-A57 64-bit ARMv8-A 2 GHz AMD Opteron A1100 SoC, 16 GB of RAM, a 512 GB SATA3 HDD, and an AMD XGBE 10 GbE NIC, running Ubuntu 16.04. For client-server experiments, the clients ran on a Lenovo ThinkPad P52 with a quad-core Intel i7-8750H 64-bit 4.1 GHz CPU, 32 GB RAM, and a 1 TB PCIe SSD, running Linux Mint 20, connected to the Arm hardware via Gigabit Ethernet through an ASUS RT-N16. All machines used Linux kernel 5.4 LTS and for running in containers, the Docker 20.10.6 container runtime.

We ran the microbenchmarks and application workloads listed in Table 3 using the following five system configurations: (1) natively on the host without containers to provide a baseline measure of performance, (2) Docker with unmodified Linux containers (Docker), (3) BlackBox running Docker with traditional Linux containers, without the security guarantees of being enclaved (BlackBox NS, for Non-Secure), (4) BlackBox running Docker with enclaved Linux containers without encrypted IPC (BlackBox NE, for no encryption), and (5) BlackBox running Docker with enclaved Linux containers (BlackBox Enclaved). Three BlackBox configurations were used to quantify the cost of different protection mechanisms. BlackBox NS provides the same security as Docker, the only difference being that BlackBox NS runs the containers on BlackBox with the OS's NPT enabled, to quantify NPT overhead. BlackBox NE provides stronger security by enclaving the container but without enabling IPC encryption, thereby quantifying BlackBox overhead without IPC encryption. BlackBox Enclaved is the same as BlackBox NE but with IPC encryption enabled. When using BlackBox, its DMA protection is not available on the Raspberry Pi 4 because it has no SMMU. Docker's default `seccomp` policy is enabled for all configurations. Versions of `libseccomp` prior to v2.5 had a significant performance issue on policies like Docker's default [65]. The Docker version we use incorporates this performance fix.

5.1 Performance Measurements

Figure 4 shows performance measurements for each microbenchmark and application workload for each container configuration normalized to native execution; lower numbers are better. Solid bars indicate results run on the Raspberry Pi and the overlaid outlined bars indicate results run on the AMD Seattle Arm server. BlackBox has the highest overhead relative to native execution on the null system call measurement, but most of the overhead is from Docker, due to its use of `seccomp` to configure and limit the system calls available in a container to reduce the available attack surface area. Although `seccomp` is used for all system calls, its overhead is most apparent for the null system call as its base cost is the lowest since it does no work. In contrast, the overhead due to BlackBox, from the two CSM calls that BlackBox makes on every system call, is small relative to `seccomp`. Although CSM calls require switching to and from Arm's EL2 mode, it requires no more than EL2's system register state to execute, eliminating the need to save and restore system registers when switching between EL1 and EL2; only general-purpose registers need to be saved and restored. Taking advantage of Arm's architectural features makes CSM calls relatively inexpensive, enabling fine-grained container protection without significant overhead from system call interposition. The key aspect of Arm's design that is crucial for the CSM is that software determines what state needs to be saved and restored. Running the CSM in the equivalent x86 hypervisor root mode would be much more expensive as it provides a hardware instruction that must be used to context switch to root mode that requires saving and restoring the entire CPU system state [17]. The x86 mechanism works well for hypervisors since they already require this operation, but poorly for the CSM which makes minimal use of CPU system state, and therefore does not need the expensive save and restore.

For the `read`, `write`, `stat`, `open/close`, and `select` system call measurements, BlackBox Enclaved is less than two times the cost of Docker. The overhead for the

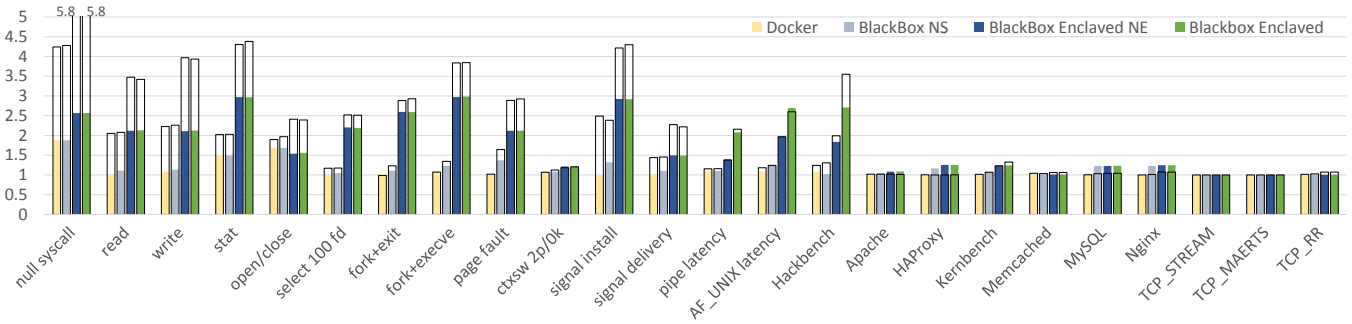


Figure 4: Container Performance for Microbenchmarks and Application Workloads.

enclaved configurations is due to the need to copy system call arguments back and forth between the container PPAS and OS, since enclaved container memory is not accessible to the OS. `open` additionally incurs overhead as part of checking the path being opened to identify FDs associated with shared libraries as part of BlackBox's binary decryption mechanism. For all system calls, the overhead on the AMD server, as indicated by the outlined bars, exceeds that of the Raspberry Pi's. In most cases, this is due to the server hardware performing the CPU bound system call operations more quickly than the Raspberry Pi while their memory performance remains similar, resulting in the similar costs for BlackBox's system call argument copying having relatively higher overhead.

`fork` and `exec` measurements show the highest overhead for BlackBox Enclaved versus Docker, less than three times the cost of Docker. This is due to validating that the new process's address space matches its parent's on `fork`, and additionally validating the address space against the new binary's mappings on `exec`. Although the binary must be decrypted for `exec` measurements, it is only decrypted once and all subsequent iterations just confirm the mappings match the first's, thereby amortizing the cost of the initial decryption.

Page fault measurements show the one microbenchmark for which there is noticeable overhead for BlackBox NS versus Docker. This is due to the added cost of using NPTs for the BlackBox NS configuration. This overhead then increases for enclaved containers due to needing to verify the fault resides within a known address mapping to protect the container from potential Iago attacks from the OS. Although a page fault results in several context switches to the CSM, the context switches themselves are not a significant cost because they are relatively inexpensive on Arm.

Protecting container IPC communication through encryption imposes little cost for most workloads, but this overhead is noticeable for pipe, UNIX domain sockets (AF_UNIX), and `hackbench` measurements. These benchmarks represent worst-case overheads for IPC encryption because they all use IPC to read and write a single byte to signal other processes. When encrypting, this single byte is padded and written along with authentication data, significantly increasing the relative write size and affecting read/write latency measurements. In contrast, the context switch microbenchmark, in which a parent process spawns two child processes that communicate between each other with pipes, has almost no overhead. In this case, 4 byte reads and writes are used so the extra data that encryption adds, and therefore the time to complete the calls, is relatively less, and context switching and rescheduling dominates IPC encryption costs. The signaling microbenchmarks do not involve any encryption. BlackBox Enclaved overhead for signal installation is due to copying the `sigaction` struct in and out, and for signal delivery is due to verifying the control flow.

Apache, HAProxy, Kernbench, memcached, MySQL, and Nginx measurements show that BlackBox overhead is much less on realistic application workloads than microbenchmarks. In most cases, BlackBox Enclaved overhead versus native execution is less than 15% on both the Raspberry Pi and AMD server, demonstrating modest overhead across both Arm embedded and server hardware. As indicated by the BlackBox NS measurements, NPT usage is a source of overhead, though more so on the Raspberry Pi than the AMD server. Apache, HAProxy, and Nginx workloads measure latency in addition to throughput. In terms of latency, the overhead for these workloads for BlackBox Enclaved versus native execution is less than 15% on both the Raspberry Pi and AMD server. Furthermore, Netperf measurements show that BlackBox provides fast networking performance as it involves no I/O virtualization, in contrast to using VMs. Applications are able to make full use of the host's networking capabilities. Although applications are expected to encrypt their network I/O to protect their data, we did not encrypt network connections for these measurements to avoid encryption costs obscuring BlackBox's overhead.

Figure 5 quantifies the CPU utilization when running the application workloads, as a measure of computational overhead. Solid bars indicate results run on the Raspberry Pi and the overlaid outlined bars indicate results run on the AMD server. CPU utilization is generally lower on the AMD server than the Raspberry Pi, since the AMD server is more powerful with more CPUs. On the Raspberry Pi, the difference in CPU utilization between BlackBox Enclaved and native execution is less than 15% across all workloads, and less than 5% for all workloads except Apache and Memcached. On the AMD server, the difference in CPU utilization between BlackBox

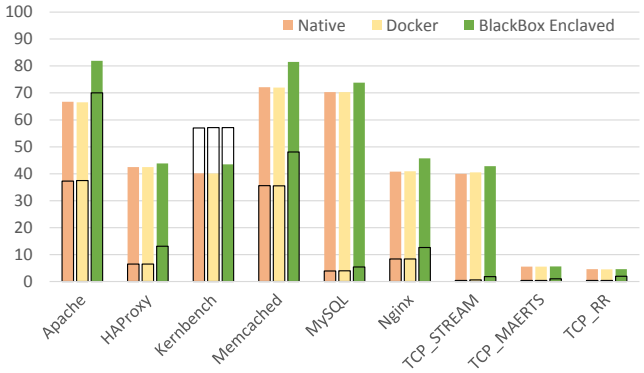


Figure 5: CPU Utilization for Application Workloads

Enclaved and native execution is less than 15% across all workloads, except Apache. Apache CPU utilization for BlackBox Enclaved is high because at higher throughput rates, the cost of extra copying to use syscall buffers, as discussed in Section 3.4, becomes dominant. The buffers are used to send data from a container's PPAS to the OS to perform network I/O. Other than Apache, the difference in CPU utilization between BlackBox Enclaved and native execution is quite modest across both Arm embedded and server hardware.

5.2 System Call Coverage

We evaluated the completeness of Linux system call support in the current BlackBox prototype implementation by running the Linux Test Project (LTP) [44] version 20210524 system call test suite. LTP consists of 1344 test cases designed to test for correct functionality across the entire Linux system call interface. We compared system call support results for running LTP in an enclaved container on BlackBox versus running it natively, in both cases using the Raspberry Pi. When running LTP natively, 1149 test cases pass and 195 fail. These failures are expected and are a combination of missing dependencies and unsupported features of the kernel and architecture used. For example, test cases for the 16-bit version of `fchown` are not supported on the platform. When running LTP using BlackBox, 1012 test cases pass and 332 fail, demonstrating support for almost 90% of test cases that passed when run natively. The additional 137 failed tests are due to the current prototype not yet supporting lesser used system calls like `process_vm_readv`.

5.3 Evaluation of Practical Attacks

We evaluated BlackBox's effectiveness against a compromised OS by analyzing CVEs related to the Linux kernel and various Linux container engines such as Docker. We considered 23 CVEs which could result in privilege escalation, code execution, and memory corruption in Linux capable of compromising the integrity and confidentiality of container data; we did not consider denial of service attacks, as BlackBox does not guarantee availability. Specifically, privilege escalation occurs if the exploit enables the attacker to gain root access or kernel privilege level, and code execution occurs if the exploit enables executing arbitrary code at the same privilege as the software with the bug.

Bug (CVE-*)	Description
2009-3234	Kernel buffer overflow enabling return-to-user attack.
2010-2959	Function pointer overwrite due to integer overflow.
2010-4258	Kernel memory overwrite due to improper handling of get_fs value.
2013-6441	Improper permissions when mounting /sbin/init.
2014-6407	Symbolic and hardlink issues during docker pull.
2014-9357	Mishandling untrusted archive extraction.
2015-1335	Directory traversal flaw in lxc-start.
2015-3627	Unchecked file descriptor opened prior to chroot.
2015-3629	Unchecked symlink when respawning container.
2015-3630	Weak permissions on /proc filesystem.
2016-1576	Improperly restricted mount namespace.
2016-5195	Race condition in handling CoW breakage.
2016-7117	Use after free in __sys_recvmmsg.
2016-9962	Improperly flushed file descriptors.
2017-7308	Improper validation of data size in packet_set_ring().
2017-1000112	Exploitable memory corruption due to UFO to non-UFO path switch.
2018-15664	TOCTOU vulnerability in symbolic link checking.
2018-18955	Mishandled nested user namespaces in map_write().
2019-5736	/proc/self/exe file descriptor mishandling
2019-10144	Container processes not isolated during 'rkt enter'.
2019-11247	Improper access to cluster-scoped custom resource.
2019-14271	Container contents loaded while privileged during container copy.
2020-14386	Kernel memory corruption due to arithmetic issue in tpacket_rcv().

Table 4: CVEs Used for Evaluation of Practical Attacks

Table 4 lists the CVEs considered. We considered both malicious containers and unprivileged host users who exploit bugs in the kernel and container engines to elevate privileges and compromise container data. In general, these CVEs exploit flaws in container runtime systems and the kernel that enable an attacker to obtain kernel-level or root-level access. Ordinarily, this level of access compromises all container data and integrity on the system. Linux and the relevant container engine do not fully protect against any of these compromises. In contrast, BlackBox protects against all of them.

6 Related Work

Various approaches have been explored to securing applications from untrusted OSes. Hardware-based trusted execution environments (TEEs) such as ARM TrustZone [2] and Intel Software Guard Extensions (SGX) [30] can protect application memory from higher privileged software, but require applications to be written or rewritten specifically for this purpose and may impose other functionality restrictions.

Some systems have built on TEEs. Haven [7] aims to en-

clave Windows applications by porting a Windows library OS to run inside SGX, avoiding Iago attacks by trusting the library OS at the cost of a significant TCB. Other systems also propose running library OSes enclaved by SGX [50,59,66]. CubicleOS [57] is a library OS designed to be runnable within containers that makes use of Intel MPK hardware extensions to isolate apps. Scone [3] uses SGX to enclave Linux containers, requiring its own custom threading model and a modified C library within SGX to provide system call support and shielded I/O interfaces for interacting with the OS. TZ-Container [28] leverages a shield layer and a container manager inside TrustZone to protect containers, but relies on the OS not modifying the memory mappings used to protect containers by scanning the OS image to ensure it does not contain instructions capable of updating page tables. TrustShadow [24] introduces a runtime system within TrustZone so that a limited number of security-critical legacy apps operate on TrustZone memory isolated from the OS. Unlike these approaches, BlackBox does not rely on TrustZone or SGX and does not rely on a library OS or other significant runtime system running inside an enclaved execution environment, avoiding increasing TCB complexity. Unlike Haven, its small TCB comes with potentially greater susceptibility to Iago attacks by allowing applications to use the system call interface of the untrusted OS.

Commodity hypervisors have been modified to secure applications from an untrusted OS by restricting a guest OS in a VM to an encrypted view of application memory [4,10,11,27,35,45,67]. For example, InkTag [27] uses two NPTs as part of its isolation mechanism, one for the OS and the other for all applications, separating the plaintext memory of isolated applications from encrypted memory, but relying on paravirtualized page table updates to isolate applications from each other. Appshield [12] uses virtualization techniques to protect and isolate critical applications against OS-level malware attacks. Appshield's memory protection model requirements are not compatible with Linux's copy-on-write semantics and its limited system call interface is insufficient to support significant workloads. In contrast, BlackBox does not rely on a hypervisor or traditional memory virtualization, but instead introduces a new concept of protected physical address spaces implemented as part of a container security monitor, enabling it to have a much smaller TCB.

Various approaches reduce the hypervisor's TCB. Microhypervisors [25,34,61] build new hypervisors from scratch with smaller TCBs, but at the cost of a significantly reduced feature set. BlackBox's approach allows for a small TCB while still maintaining a significant feature set and the full hardware support available in a commodity OS. SeKVM [38–40,63] retrofits KVM with a small verified TCB to provide VM data confidentiality and integrity. In contrast, BlackBox provides container-level isolation and does not require a hypervisor, introducing a new concept, the CSM, that avoids the cost and complexity of hypervisor-based virtualization.

X-Containers [60] targets securely isolating containers in the cloud. Its containers include an entire library OS based on Linux and run on top of a Xen hypervisor, providing a model more akin to nested virtualization. Unlike BlackBox, X-Containers have a large TCB from requiring both large library OSes and a commodity hypervisor.

Other approaches have looked at ways to harden traditional containers. gVisor [23] runs a limited userspace kernel within a container and beneath applications. System calls are intercepted to further isolate applications from the host OS through reduced interactions and potential attack surfaces. gVisor's increased isolation comes at the cost of a increased TCB size in the container. Distroless images [22] aim to limit the contents of a container to precisely what is necessary for the target app to run, reducing what must be trusted and maintained within a container. Linux Container Hardening [42] aims to improve the security of Linux containers through improving the kernel subsystems and primitives used by containers to be more secure. These approaches are complementary to BlackBox, and although they improve container security, unlike BlackBox, they all must still trust the OS and its large codebase.

7 Conclusions

BlackBox is a new container architecture providing fine-grain protection of application data confidentiality and integrity without trusting the OS. BlackBox achieves this by introducing a container security monitor, a new software component that creates protected physical address spaces for containers. The monitor enforces protected address spaces to isolate container memory and CPU state from the OS and other containers. It facilitates the use of OS facilities via system calls by passing required data between protected address spaces and the OS, implicitly declassifying such data. This narrow purpose keeps it small and simple. Unlike a hypervisor, the monitor performs no virtualization or resource management. Instead, it relies on the OS to provide complex functionality required to manage hardware resources, including CPU scheduling, memory management, file systems, and device management. We have implemented BlackBox by repurposing Arm hardware virtualization support. Our results demonstrate that BlackBox supports existing unmodified containerized application workloads with modest overhead while maintaining a trusted computing base orders of magnitude less than an OS or commodity hypervisor.

8 Acknowledgments

Shih-Wei Li and Xuheng Li helped with system implementation. Andrew Baumann, Christoffer Dall, Peter Pietzuch, and Nicolas Viennot provided helpful comments on earlier drafts. This work was supported in part by OPPO, a Guggenheim Fellowship, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080.

References

[1] Amazon Web Services, Inc. AWS Nitro Enclaves User Guide. https://docs.aws.amazon.com/enclaves/latest/user/building-eif.html, May 2022.

[2] ARM Ltd. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, April 2009.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Linda, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 2016)*, pages 689–703, Savannah, GA, November 2016.

[4] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. SICE: A Hardware-Level Strongly Isolated Computing Environment for X86 Multi-Core Platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 375–388, Chicago, IL, October 2011.

[5] Michael Backes, Goran Doychev, and Boris Kopf. Preventing Side-Channel Leaks in Web Traffic: A Formal Approach. In *Proceedinsgs of the 20th ISOC Network and Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, February 2013.

[6] Ricardo Baratto, Shaya Potter, Gong Su, and Jason Nieh. MobiDesk: Mobile Virtual Desktop Computing. In *Proceedings of the 10th Annual ACM International Conference on Mobile Computing and Networking (MobiCom 2004)*, pages 1–15, Philadelphia, PA, September 2004.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI 2014)*, pages 267–283, Broomfield, CO, October 2014.

[8] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, February 2017.

[9] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 253–264, Houston, TX, March 2013.

[10] Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang, Pen chung Yew, and Wenbo Mao. Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor. Technical Report PPITR-2007-08001, Parallel Processing Institute, Fudan University, August 2007.

[11] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, pages 2–13, Seattle, WA, March 2008.

[12] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. Efficient Virtualization-Based Application Protection Against Untrusted Operating System. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2015)*, pages 345–356, Singapore, Republic of Singapore, April 2015.

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

[14] Rongzhen Cui, Lianying Zhao, and David Lie. Emilia: Catching Iago in Legacy Code. In *Proceedings of the 2021 ISOC Network and Distributed Systems Security Symposium (NDSS 2021)*, Virtual Event, February 2021.

[15] Christoffer Dall, Jeremy Andrus, Alex Van't Hof, Oren Laadan, and Jason Nieh. The Design, Implementation, and Evaluation of Cells: A Virtual Mobile Smartphone Architecture. *ACM Transactions on Computer Systems (TOCS)*, 30(3):9:1–31, August 2012.

[16] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, and Jason Nieh. ARM Virtualization: Performance and Architectural Implications. *ACM SIGOPS Operating Systems Review*, 52(1):45–56, July 2018.

[17] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, pages 304–316, Seoul, South Korea, June 2016.

[18] Christoffer Dall, Shih-Wei Li, and Jason Nieh. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, pages 221–234, Santa Clara, CA, July 2017.

[19] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 333–347, Salt Lake City, UT, March 2014.

[20] Docker, Inc. Empowering App Development for Developers - Docker. https://www.docker.com, 2021.

[21] Google. HTTPS Encryption on the Web – Google Transparency Report. https://transparencyreport.google.com/https/overview, April 2018.

[22] Google, Inc. "Distroless" Container Images. https://github.com/GoogleContainerTools/distroless, February 2022.

[23] Google, Inc. gVisor: Application Kernel for Containers. https://github.com/google/gvisor, May 2022.

[24] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In *Proceedings of the 15th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2017)*, pages 488–501, Niagara Falls, NY, June 2017.

[25] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Systems (APSys 2010)*, pages 19–24, New Delhi, India, August 2010.

[26] Alexander Van't Hof and Jason Nieh. AnDrone: Virtual Drone Computing in the Cloud. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 2019)*, pages 6:1–16, Dresden, Germany, March 2019.

[27] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 265–278, Houston, TX, March 2013.

[28] Zhichao Hua, Yang Yu, Jinyu Gu, Yubin Xia, Haibo Chen, and Binyu Zang. TZ-Container: Protecting Container From Untrusted OS with ARM TrustZone. *Science China Information Sciences*, 64:1869–1919, August 2021.

[29] Solomon Hykes. Introducing runC: A lightweight Universal Container Runtime. https://www.docker.com/blog/runc/, June 2015.

[30] Intel Corporation. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, October 2014.

[31] International Organization for Standardization and International Electrotechnical Commission. ISO/IEC 11889-1:2015 - Information technology – Trusted Platform Module Library. https://www.iso.org/standard/66510.html, September 2016.

[32] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P 2015)*, pages 591–604, San Jose, CA, May 2015.

[33] Rick Jones. Netperf. https://github.com/HewlettPackard/netperf, June 2018.

[34] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2019)*, pages 207–220, Big Sky, MT, October 2009.

[35] Youngjin Kwon, Alan M. Dunn, Michael Z. Lee, Owen S. Hofmann, Yuanzhong Xu, and Emmett Witchel. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016)*, pages 277–290, Atlanta, GA, 2016.

[36] Susan Landau. Making Sense from Snowden: What's Significant in the NSA Surveillance Revelations. *IEEE Security and Privacy*, 11(4):54–63, July 2013.

[37] Let's Encrypt. Let's Encrypt Stats - Let's Encrypt. https://letsencrypt.org/stats/, April 2018.

[38] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1357–1374, Santa Clara, CA, August 2019.

[39] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (IEEE S&P 2021)*, pages 1782–1799, San Francisco, CA, May 2021.

[40] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, pages 3953–3970, Vancouver, BC Canada, August 2021.

[41] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, Carlsbad, CA, July 2022.

[42] Linux Container Hardening Project. Linux Container Hardening. https://containerhardening.org/, 2021.

[43] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P 2015)*, pages 605–622, San Jose, CA, May 2015.

[44] LTP developers. LTP - Linux Test Project. https://linux-test-project.github.io/, 2021.

[45] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (IEEE S&P 2010)*, pages 143–158, May 2010.

[46] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (USENIX ATC 1996)*, pages 279–294, San Diego, CA, January 1996.

[47] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 361–376, Boston, MA, December 2002.

[48] Shaya Potter and Jason Nieh. Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 2010)*, pages 103–116, Boston, MA, June 2010.

[49] Shaya Potter and Jason Nieh. Improving Virtual Appliance Management through Virtual Layered File Systems. In *Proceedings of the 25th Large Installation System Administration Conference (LISA 2011)*, pages 25–38, Boston, MA, December 2011.

[50] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter R. Pietzuch. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *ArXiv*, abs/1908.11143, January 2020.

[51] Redis Labs. memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark, April 2015.

[52] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 199–212, Chicago, IL, November 2009.

[53] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC))*, 15(1), March 2012.

[54] Rusty Russell. Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, January 2008.

[55] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, November 1984.

[56] Samsung Electronics Co., Ltd. Samsung Knox - White Paper. https://docs.samsungknox.com/admin/whitepaper/kpe/samsung-knox.htm, 2021.

[57] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*, pages 546–558, Virtual Event, April 2021.

[58] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 552–561, Alexandria, VA, October 2007.

[59] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*, pages 955–970, Lausanne, Switzerland, March 2020.

[60] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-Containers: Breaking Down Barriers to Improve Performance and Isolation

of Cloud-Native Containers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, pages 121–135, Providence, RI, April 2019.

[61] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010)*, pages 209–222, Paris, France, April 2010.

[62] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2012)*, pages 21–41, Heraklion, Crete, Greece, July 2013.

[63] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP 2021)*, pages 866–881, Virtual Event, Germany, October 2021.

[64] The Apache Software Foundation. ab - Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs/2.4/programs/ab.html, April 2015.

[65] Tom Hromatka. RFE: Use a cBPF Binary Tree for Large Seccomp Filters. https://github.com/seccomp/libseccomp/issues/116, 2018.

[66] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, pages 645–658, Santa Clara, CA, July 2017.

[67] Jisoo Yang and Kang G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, pages 71–80, Seattle, WA, March 2008.

[68] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, pages 305–316, Raleigh, NC, October 2012.

[69] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in Paas Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*, pages 990–1003, Scottsdale, AZ, November 2014.

[70] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 1789–1806, Dallas, TX, October 2017.

Blockaid: Data Access Policy Enforcement for Web Applications

Wen Zhang[1] Eric Sheng[2,*] Michael Chang[1] Aurojit Panda[3] Mooly Sagiv[4] Scott Shenker[1,5]

[1]*UC Berkeley* [2]*Yugabyte* [3]*NYU* [4]*Tel Aviv University* [5]*ICSI*

Abstract

Modern web applications serve large amounts of sensitive user data, access to which is typically governed by data-access policies. Enforcing such policies is crucial to preventing improper data access, and prior work has proposed many enforcement mechanisms. However, these prior methods either alter application semantics or require adopting a new programming model; the former can result in unexpected application behavior, while the latter cannot be used with existing web frameworks.

Blockaid is an access-policy enforcement system that preserves application semantics and is compatible with existing web frameworks. It intercepts database queries from the application, attempts to verify that each query is policy-compliant, and blocks queries that are not. It verifies policy compliance using SMT solvers and generalizes and caches previous compliance decisions for better performance. We show that Blockaid supports existing web applications while requiring minimal code changes and adding only modest overheads.

1 Introduction

Many modern web applications use relational databases to store sensitive user data, access to which is governed by organizational or regulatory *data-access policies*. To enforce these policies, today's web developers wrap each database query within access checks that determine whether a user has access to the queried data. As an application can query the database at many call sites, getting access checks right at every call site is challenging, and erroneous or missing checks have exposed sensitive data in many production systems [4, 30, 37, 38, 47, 64].

Prior work has suggested a variety of languages, frameworks, and tools that simplify the enforcement of data-access policies. As we detail in §2, these approaches either (1) require applications be written using specialized web frameworks, hindering their adoption; or (2) transparently remove from query results any data that cannot be revealed, possibly resulting in unexpected application behavior (e.g., the user has no idea that there are missing results and reaches the wrong conclusion).

[*]Work done while at UC Berkeley.

This paper proposes an alternative approach to enforcing data-access policies that meets four goals:

1. **Backwards compatibility**: Applies to applications built using common existing web frameworks.
2. **Semantic transparency**: Fully answers queries that comply with the policy and blocks queries that do not (rather than providing partial, and potentially misleading, results).
3. **Policy expressiveness**: Supports a wide range of policies.
4. **Low overhead**: Has limited impact on page load time.

We implement this approach in Blockaid, a system that enforces a data-access policy at runtime by intercepting SQL queries issued by the application, verifying that they comply with the policy, and blocking those that do not. We assume non-compliant queries are rare in production (having been mostly eliminated in testing), and focus on efficiently checking compliant queries. Blockaid expects the developer to insert access checks as usual; it merely ensures that the checks are adequate.

A Blockaid policy consists of SQL view definitions that specify what information can be accessed by a given user, although the application still issues queries against the base tables as usual (rather than against the views). Under this setting, a query is compliant if it *never* reveals—for any underlying dataset—more information than the views do, a well-studied property in databases called *query determinacy* [51].

While determinacy characterizes the compliance of one query in isolation, it is too restrictive in the context of web applications, which typically issue multiple queries when serving a request. In this setting, what queries can be allowed often depends on the result of previous queries in the same web request. Thus, we extend determinacy to take a *trace* of previous queries and their responses, a novel extension we call *trace determinacy*, and use that as the criterion for compliance.

To verify compliance, Blockaid frames trace determinacy as an SMT formula and checks it using SMT solvers. As we later explain, a solver returns an unsatisfiability proof when a query is compliant, and a test demonstrating a violation otherwise.

This basic method, while correct, is impractically slow as it invokes solvers on every query. Thus, we use a *decision cache* to record compliant queries (with traces) so that future

occurrences need not be rechecked. But caching *exact* queries and traces would be ineffective: a query is usually specific to the user and page visited, and so is unlikely to occur many times.

Thus, to increase cache hit rate, we implement a novel generalization mechanism which, given a compliant query-trace pair, extracts a small set of assumptions on the query and trace that alone would guarantee compliance. These assumptions are cached in the form of a *decision template*, which will apply to all future query-trace pairs that meet those assumptions. Blockaid generates decision templates by progressively relaxing a query and trace while maintaining compliance, with the help of solver-generated unsat cores [8, § 11.8]. It does not cache noncompliance results, which we expect to be rare in production as they typically indicate application/policy bugs.

We applied Blockaid to three existing applications—diaspora* [25], Spree [63], and Autolab [5]—and found that it imposes an overhead of 2 % to 12 % to the median page load time when compliance decisions are cached.

Blockaid has some important limitations. It assumes that the application obtains all of its information through SQL queries visible to Blockaid or from a caching layer or file system mediated by Blockaid. It also supports only a subset of SQL and is at the mercy of solver performance and unsat-core size.

Blockaid is open source at `https://github.com/blockaid-project`, and further theoretical discussions can be found in the appendices of our extended technical report [73].

2 Related Work

The subject of data-access control has been studied by many. We compare our approach to prior ones along our goals (§1).

Static verification. Several systems have been proposed to statically verify that application code can only issue compliant queries; examples include Swift [17], SELINKS [22], UrFlow [15], and STORM [42]. These systems incur no run-time overhead and can be more precise than Blockaid as they analyze source code. However, they typically require using a specialized language or framework like Jif [50] or Ur/Web [16], sacrificing compatibility with common web frameworks.

Query modification. A popular run-time approach is query modification [65]: replacing secret values returned by a query with placeholders (or dropping any rows containing secrets). This is implemented in commercial databases [13, 49] and academic works like Hippocratic databases [3], Jacqueline [72], Qapla [48], and multiverse databases [46]. While this approach allows programmers to issue queries without regard to policies, it lacks semantic transparency as it can alter query semantics in unexpected ways and return misleading results [32, 58, 69].

Furthermore, many of the query modification mechanisms use row- and cell-level policies (e.g., SQL Server RLS and DDM, Oracle VPD). As we discuss in §9, this row/cell-level format is less expressive than Blockaid's view-based scheme.

View-based access control. Many databases allow creating views and granting access to views and tables. Although identical in expressiveness to Blockaid, this mechanism requires queries to explicitly use view names instead of table names (like `Users`). This marks a significant deviation from regular web programming, as programmers must now sort out which views to use for each query. In contrast, Blockaid allows queries to be issued against the base tables directly.

While some prior work has studied view-based compliance of queries issued against base tables [10, 11], they only check single queries, while Blockaid checks a query in the context of a trace, a crucial feature for supporting web applications.

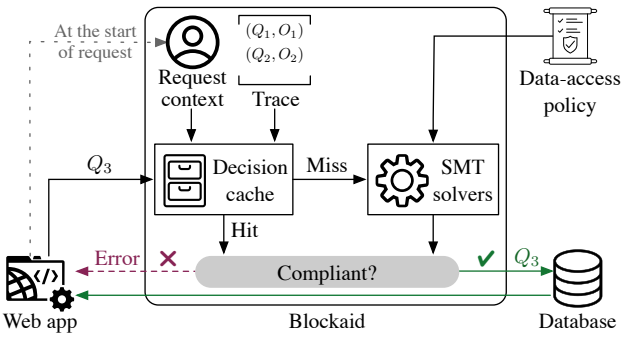


Figure 1: *An overview of Blockaid (for a single web request).*

3 System Design

3.1 Application Assumptions and Threat Model

Blockaid targets web applications that store data in a SQL database. We assume that a user is logged in and that the current user's identifier is stored in a *request context*. The application can access the database and the request context when serving a request; each request is handled independently from others. We assume that the application authenticates the user correctly, and that the correct request context is passed to Blockaid (§3.2).

A *data-access policy* dictates, for a given request context, what information in the database is *accessible* and what is *inaccessible*. We treat the database schema and the policy itself as public knowledge and assume that the user cannot use side channels to circumvent policies. We enforce policies on database *reads* only, as done in prior work [2, 10–12, 33, 41, 46, 58, 61, 65, 69]. Ensuring the integrity of updates, while important, is orthogonal to our goal and is left to future work.

3.2 System Overview

Blockaid is a SQL proxy that sits between the application and the database (Figure 1). It takes as input (1) a database schema (including constraints), and (2) a data-access policy specified as database views (§4), and checks query compliance for each web request separately. For each web request, it maintains a *trace* of queries issued so far and their results; the trace is cleared when the request ends. Blockaid assumes that the results returned by queries in the trace are not altered till the end of the request.

When a web request starts, the application sends its request context to Blockaid. Then, every SQL query from the application traverses Blockaid, which attempts to verify that the query

is *compliant*—i.e., it can be answered using accessible information only. To do so, Blockaid checks the decision cache for any similar query has been determined compliant previously. If not, it encodes noncompliance as an SMT formula (§5) and checks its satisfiability using several SMT solvers in parallel (§7).

If a query is compliant, Blockaid forwards it to the database unmodified. In case of a cache miss, Blockaid also extracts and caches a decision template (§6). Finally, it appends the query and its result to the trace. If verification fails, Blockaid blocks the query by raising an error to the application.

Although our core design assumes that all sensitive information is stored in the relational database, Blockaid supports limited compliance checking for two other common data sources:

1. If the application stores database-derived data in a **caching layer** (e.g., Redis), the programmer can annotate a cache key pattern with SQL queries from which the value can be derived. Blockaid can then intercept each cache read and verify the compliance of the queries associated with the key.
2. If the application stores sensitive data in the **file system**, it can generate hard-to-guess names for these files and store the file names in a database column protected by the policy.

Blockaid's basic requirement is soundness: preventing the revelation of inaccessible information (formalized in §4.3). However, it may reject certain behaviors that do not violate the policy (§9), although such false rejections never arose in our evaluation (§8).

We end by emphasizing two aspects of Blockaid's operation:

1. Blockaid has *no visibility into or control over* the application (except by blocking queries). So it must assume that *any* data fetched by the application will be shown to the user.
2. Blockaid has *no access* to the database except by observing query results—it cannot issue additional queries of its own.

3.3 Application Requirements

For use with Blockaid, an application must:

1. Send the request context to Blockaid at the start of a request and signal Blockaid to clear the trace at the end;
2. Handle rejected queries cleanly (although a web server's default behavior of returning HTTP 500 often suffices); and,
3. Not query data that it does not plan on revealing to the user.

Existing applications often violate the third requirement. For example, when a user views an order on a Spree e-commerce site, the order is fetched from the database, and only then does Spree check, in application code, that the user is allowed to view it. To avoid spurious errors from Blockaid, such applications must be modified to fetch only data known to be accessible.

4 View-based Policy and Compliance

Throughout the paper, we will use as a running example a calendar application with the following database schema:

$$Users(\underline{UId}, Name)$$
$$Events(\underline{EId}, Title, Duration)$$
$$Attendances(\underline{UId}, \underline{EId}, ConfirmedAt)$$

Listing 1: *Example policy view definitions V_1 to V_4 for the calendar application.* `?MyUId` *refers to the current user ID.*

1. `SELECT * FROM Users`
   Each user can view the information on all users.
2. `SELECT * FROM Attendances WHERE UId = ?MyUId`
   Each user can view their own attendance information.
3. `SELECT * FROM Events WHERE EId IN (SELECT EId FROM Attendances WHERE UId = ?MyUId)`
   Each user can view the information on events they attend.
4. `SELECT * FROM Attendances WHERE EId IN (SELECT EId FROM Attendances WHERE UId = ?MyUId)`
   Each user can view all attendees of the events they attend.

where primary keys are underlined. The request context consists of a parameter *MyUId* denoting the *UId* of the current user.

4.1 Specifying Policies as Views

A policy is a collection of SQL queries that, together, define what information a user is allowed to access. Each query is called a *view definition* and can refer to parameters from the request context. As an example, Listing 1 shows four view definitions, V_1–V_4; we denote this policy as $\mathcal{V} = \{V_1, V_2, V_3, V_4\}$.

Notationally, for a view V and a request context ctx, we write V^{ctx} to denote V with its parameters replaced with values in ctx. We often drop the superscript when the context is apparent.

4.2 Compliance to View-based Policy

Under a policy consisting of view definitions, Blockaid can allow an application query to go through *only if* it is certain that the query's result is *uniquely determined* by the views. In other words, an allowable query must be answerable using accessible information alone. If a query's output *might* depend on information outside the views, Blockaid must block the query.

Example 4.1. Let $MyUId = 2$. The following query selects the names of everyone whom the user attends an event with:

```
SELECT DISTINCT u.Name
FROM   Users u
       JOIN Attendances a_other
         ON a_other.UId = u.UId
       JOIN Attendances a_me
         ON a_me.EId = a_other.EId
WHERE a_me.UId = 2
```

Looking at Listing 1, this query can always be answered by combining V_4, which reveals the *UId* of everyone whom the user attends an event with, with V_1, which supplies the names associated with these *UId*'s. Hence, Blockaid allows it through.

This query above is allowed *unconditionally* because it is answerable using the views on *any* database instance. More commonly, queries are allowed *conditionally* based on what Blockaid has learned about the current database state, given the trace of prior queries and results in the same web request.

Example 4.2. Again, let $MyUId = 2$. Consider the following sequence of queries issued while handling one web request:

```
1. SELECT * FROM Attendances
   WHERE UId = 2 AND EId = 5
   ↪ (UId=2, EId=5, ConfirmedAt="05/04 1pm")
2. SELECT Title FROM Events WHERE EId = 5
```

The application first queries the user's attendance record for Event #5—an unconditionally allowed query—and receives one row, indicating the user is an attendee. It then queries the title of said event. This is allowed because V_3 reveals the information on all events attended by the user. More precisely, the trace limits our scope to only databases where the user attends Event #5. Because the second query is answerable using V_3 *on all such databases*, it is conditionally allowed given the trace.

Context is important here: the second query cannot be safely allowed if it were issued in isolation.

Example 4.3. Suppose instead that the application issues the following query by itself:

```
SELECT Title FROM Events WHERE EId = 5
```

Blockaid must block this query because it is not answerable using $\mathcal{V}$ on a database where the user does not attend Event #5. Whether or not the user *actually* is an attendee of the event is irrelevant: The application, not having queried the user's attendance records, cannot be certain that the query is answerable using accessible information alone. This differs from alternative security definitions [32, 39, 74] where a policy enforcer can allow a query after inspecting additional information in the database that has not been fetched by the application.

Definition 4.4. A *trace* $\mathcal{T}$ is a sequence $(Q_1, O_1), \ldots, (Q_n, O_n)$ where each Q_i is a query and each O_i is a collection of tuples.

Such a trace denotes that the application has issued queries $Q_1, \ldots, Q_n$ and received results $O_1, \ldots, O_n$ from the database.

We now motivate the formal definition of query compliance given a trace (using colors to show correspondence between text and equations). Consider any two databases that are:

- Equivalent in terms of accessible data (i.e., they differ only in information outside the views), and
- Consistent with the observed trace (i.e., we consider only databases that *could* be the one the application is querying).

Blockaid must ensure that such two databases are indistinguishable to the user—by allowing only queries that produce the same result on both databases.

Definition 4.5. Let *ctx* be a request context, $\mathcal{V}$ be a set of views, and $\mathcal{T} = \{(Q_i, O_i)\}_{i=1}^{n}$ be a trace. A query Q is *ctx-compliant* to $\mathcal{V}$ given $\mathcal{T}$ if for every pair of databases D_1, D_2 that conform to the database schema and constraints,[1] and satisfy:

$$V^{ctx}(D_1) = V^{ctx}(D_2), \qquad (\forall V \in \mathcal{V}) \tag{1}$$

$$Q_i(D_1) = O_i, \qquad (\forall 1 \le i \le n) \tag{2}$$

$$Q_i(D_2) = O_i, \qquad (\forall 1 \le i \le n) \tag{3}$$

we have $Q(D_1) = Q(D_2)$. We will simply say *compliant* if the context is clear.

We call Definition 4.5 *trace determinacy* because it extends the classic notion of query determinacy [51, 60] with the trace. Query determinacy is undecidable even for conjunctive views and queries [27, 28]; trace determinacy must also be undecidable in the same scenario. Although several decidable cases have been discovered for query determinacy [1, 51, 53], they are not expressive enough for our use case. A promising direction is to identify classes of views and queries that capture common web use cases and for which trace determinacy is decidable.

4.3 From Query Compliance to Noninterference

Blockaid's end goal is to ensure that an application's output depends only on information accessible to the user. In relation to this goal, query compliance (Definition 4.5) satisfies two properties, making it the right criterion for Blockaid to enforce:

1. **Sufficiency**: As long as *only compliant queries* from the application are let through, there is no way for an execution's outcome to be influenced by inaccessible information.
2. **Necessity**: Any enforcement system that makes per-query decisions based solely on the query and its preceding trace *cannot safely allow any non-compliant query* without the risk of the application revealing inaccessible information.

Before stating and proving these properties formally, let us first model our target applications, enforcement systems, and goals.

We model a web request handler as a program $\mathcal{P}(ctx, req, D)$ that maps a request context *ctx*, an HTTP request *req*, and a database D to an HTTP response.[2] A program that abides by a policy $\mathcal{V}$ satisfies a *noninterference* property [21, 29] stating that its output depends only on the inputs that the user has access to—namely, *ctx*, *req*, and $V^{ctx}(D)$ for each $V \in \mathcal{V}$. The formal definition follows from a similar intuition as Definition 4.5.

Definition 4.6. A program $\mathcal{P}$ satisfies *noninterference* under policy $\mathcal{V}$ if the following condition holds:

$$\mathrm{NI}_{\mathcal{V}}(\mathcal{P}) := \forall ctx, req, D_1, D_2.\ \big[\forall V \in \mathcal{V}.\, V^{ctx}(D_1) = V^{ctx}(D_2)\big] \implies \mathcal{P}(ctx, req, D_1) = \mathcal{P}(ctx, req, D_2).$$

An enforcement system must ensure that any program running under it satisfies noninterference. We now model such a system that operates under Blockaid's assumptions.

[1] We will henceforth use "schema" to mean both schema and constraints, and rely on the database and/or the web framework to enforce the constraints.

[2] For simplicity, we assume $\mathcal{P}$ is a pure function—deterministic, terminating, and side-effect free—although this assumption can be relaxed through standard means from information-flow control [34, § 2].

Definition 4.7. An *enforcement predicate* is a mapping from a request context, a query, and a trace to an allow/block decision:

$$E(ctx, Q, \mathcal{T}) \to \{\checkmark, \times\}.$$

Definition 4.8. Let $\mathcal{P}(ctx, req, D)$ be a program and E be an enforcement predicate. We define the program $\mathcal{P}$ *under enforcement using* E as a new program $\mathcal{P}^E(ctx, req, D)$ that simulates every step taken by the original program $\mathcal{P}$, except that it maintains a trace $\mathcal{T}$ and blocks any query Q issued by $\mathcal{P}$ where $E(ctx, Q, \mathcal{T}) = \times$ by immediately returning an error.

Note that $\mathcal{P}^E$ evaluates E only on traces in which every query has been previously allowed by E given its trace prefix.

Definition 4.9. Given a request context ctx, we say that a trace $\mathcal{T} = \{(Q_i, O_i)\}_{i=1}^n$ is *prefix* E*-allowed* if for all $1 \le i \le n$,

$$E(ctx, Q_i, \mathcal{T}[1..i-1]) = \checkmark.$$

Definition 4.10. A predicate E *correctly enforces* policy $\mathcal{V}$ if:

$$\forall \mathcal{P}.\ \mathrm{NI}_{\mathcal{V}}(\mathcal{P}^E).$$

We are ready to state the sufficiency-and-necessity theorem, whose proof is left to our extended paper [73, § B]. Like before, we use colors to link a statement to its explanation.

Theorem 4.11. *Let $\mathcal{V}$ be a set of views and E be a predicate.*

1. *Suppose $E(ctx, Q, \mathcal{T}) = \checkmark$ only when Q is ctx-compliant to $\mathcal{V}$ given $\mathcal{T}$. Then E correctly enforces $\mathcal{V}$.*
2. *Suppose E correctly enforces $\mathcal{V}$. Then for any request context ctx, query Q, and prefix E-allowed trace $\mathcal{T}$ such that $E(ctx, Q, \mathcal{T}) = \checkmark$, Q is ctx-compliant to $\mathcal{V}$ given $\mathcal{T}$.*

To unpack, Theorem 4.11 says: (1) as long as an enforcement predicate ensures query compliance, it correctly enforces the policy on applications (i.e., sufficiency); and (2) for a predicate to correctly enforce the policy, it must ensure query compliance (i.e., necessity). Thus, query compliance can be regarded as the "projection" of application noninterference onto Blockaid's lens, making it the ideal criterion to enforce.

5 Compliance Checking with SMT

Having defined view-based policy and compliance, we now introduce how Blockaid verifies compliance using SMT solvers.

5.1 Translating Noncompliance to SMT

Blockaid verifies query compliance by framing *noncompliance* (i.e, the negation of Definition 4.5) as an SMT formula and checking its satisfiability—a query is compliant if and only if the formula is *unsatisfiable*. We use a straightforward translation based on Codd's theorem [20], which states, informally, that relational algebra under set semantics is equivalent in expressiveness to first-order logic (FOL). Relational algebra has five operators—projection, selection, cross product, union, and difference—and tables are interpreted as *sets* of rows (i.e., no duplicates). Under this equivalence, tables are translated to predicates in FOL, and operators are implemented using existential quantifiers, conjunctions, disjunctions, and negations.

Example 5.1. Let us translate into FOL the following query Q executed on a database D:

```
SELECT e.EId, e.Title
FROM   Events e, Attendances a
WHERE  e.EId = a.EId AND a.UId = 2
```

Let $E^D(\cdot,\cdot,\cdot)$ and $A^D(\cdot,\cdot,\cdot)$ be FOL predicates representing the *Events* and *Attendances* table in the database D in:

$$\mathsf{Q}^D(\mathsf{x}_e, \mathsf{x}_t) := \exists x_d, x_u, x'_e, x_c.\, E^D(\mathsf{x}_e, \mathsf{x}_t, x_d) \wedge A^D(x_u, x'_e, x_c) \wedge \mathsf{x}_e = x'_e \wedge x_u = 2.$$

$\mathsf{Q}^D(\mathsf{x}_e, \mathsf{x}_t)$ encodes the statement $(\mathsf{x}_e, \mathsf{x}_t) \in Q(D)$, i.e., that the row $(\mathsf{x}_e, \mathsf{x}_t)$ is returned by Q on database D. Note that Q^D is not a logical symbol, but merely a shorthand for the right-hand side.

Example 5.2. We now present the noncompliance formula for a single query Q with respect to $\mathcal{V}$ from §4.1. Let $\mathsf{V}_1^{D_i}, \ldots, \mathsf{V}_4^{D_i}$ and Q^{D_i} encode the views and query on database D_i ($i = 1, 2$) in FOL. The desired formula would then be the conjunction of:

$$\begin{aligned}
&\forall \mathbf{x}.\, \mathsf{V}_1^{D_1}(\mathbf{x}) \leftrightarrow \mathsf{V}_1^{D_2}(\mathbf{x}), && (V_1(D_1) = V_1(D_2)) \\
&\qquad\vdots \\
&\forall \mathbf{x}.\, \mathsf{V}_4^{D_1}(\mathbf{x}) \leftrightarrow \mathsf{V}_4^{D_2}(\mathbf{x}), && (V_4(D_1) = V_4(D_2)) \\
&\exists \mathbf{x}.\, \mathsf{Q}^{D_1}(\mathbf{x}) \not\leftrightarrow \mathsf{Q}^{D_2}(\mathbf{x}), && (Q(D_1) \neq Q(D_2))
\end{aligned}$$

where $\mathbf{x}$ denotes a sequence of fresh variables. Database constraints and consistency with a trace can be encoded similarly.

5.2 Handling Practical SQL Queries

The encoding of relational algebra into logic, while straightforward, fails to cover real-world SQL due to two semantic gaps:

1. While the encoding assumes that relational algebra is evaluated under *set semantics*, in practice databases use a mix of set, bag, and other semantics when evaluating queries.[3]
2. SQL operations like aggregation and sorting have no corresponding operators in relational algebra.

For Blockaid to bridge these gaps, it must first assume that database tables contain no duplicate rows. This is generally the case for web applications as object-relational mapping libraries like Active Record [59] and Django [26] add a primary key for every table. Given this assumption, Blockaid rewrites complex SQL into *basic queries* that map directly to relational algebra.

5.2.1 Basic SQL Queries

Definition 5.3. A *basic* query is either a `SELECT-FROM-WHERE` query that never returns duplicate rows, or a `UNION` of `SELECT-FROM-WHERE` clauses (the `UNION` always removes duplicates).[4]

A basic query on duplicate-free tables maps to relational algebra under set semantics, and so can be directly translated to FOL. To ensure a `SELECT` query is basic, we check it against these sufficient conditions for returning no duplicate rows:

[3] For example, a SQL `SELECT` clause can return duplicate rows, but the `UNION` operator removes duplicates.

[4] The `MINUS` operator is not used in our applications and is omitted.

- It contains the `DISTINCT` keyword or ends in `LIMIT 1`; or
- It projects unique key column(s) from every table in `FROM`, e.g., **`SELECT`** `UId, Name` **`FROM`** `Users`; or
- It is constrained by uniqueness in its `WHERE` clause—e.g.:

```
SELECT e.EId
FROM   Events e, Attendances a
WHERE  e.EId = a.EId AND a.UId = 2
```

For this query to return multiple copies of x, the database must contain multiple rows of the form $Attendances(2, x, ?)$; this is ruled out by the uniqueness constraint on (UId, EId).

In our experience, policy views can typically be written as basic queries directly—e.g., for Listing 1 we can frame V_3 and V_4 as equivalent basic queries by replacing subqueries with joins and using the inner join transformation from §5.2.2.

5.2.2 Rewriting Into Basic Queries

When the application issues a query Q, Blockaid attempts to rewrite it into a basic query Q' and verify its compliance instead. Ideally, Q' would be equivalent to Q, but when this is not possible, Blockaid produces an *approximate* Q' that reveals *at least as much* information as Q does.[5] Such approximation preserves soundness but may sacrifice completeness, although it caused no false rejections in our evaluation. We now explain how to rewrite several types of queries encountered in practice.

Inner joins. A query of the form:

```
SELECT ... FROM R1
INNER JOIN R2 ON C1 WHERE C2
```

is equivalently rewritten as the basic query:

```
SELECT ... FROM R1, R2 WHERE C1 AND C2
```

Left joins on a foreign key. Consider a query of the form:

```
SELECT ... FROM R1
LEFT JOIN R2 ON R1.A = R2.B WHERE ...
```

If `R1.A` is a foreign key into `R2.B`, then every row in `R1` matches at least one row in `R2`. In this case, the left join can be equivalently written as an inner join, which is handled as above.

Order-by and limit. Blockaid adds any `ORDER BY` column as an output column and then discards the `ORDER BY` clause. It also discards any `LIMIT` clause but, when adding the query to the trace, uses a modified condition $O_i \subseteq D(Q_i)$ (instead of "=") to indicate that it may have observed a partial result.

Aggregations. Blockaid turns **`SELECT SUM`**`(A)` **`FROM`** `R` into **`SELECT`** `PK, A` **`FROM`** `R`, where `PK` is table `R`'s primary key. By projecting the primary key in addition to `A`, the rewritten query reveals the multiplicity of the values in `A`—necessary for computing **`SUM`**`(A)`—without returning duplicate rows.

Left joins that project one table. Left joins of the form:

```
SELECT DISTINCT A.* FROM A
LEFT JOIN B ON C1 WHERE C2
```

can be equivalently rewritten to the basic query:

```
(SELECT A.* FROM A
 INNER JOIN B ON C1 WHERE C2)
UNION
(SELECT * FROM A WHERE C3)
```

where `C3` is obtained by replacing each occurrence of `B.?` with `NULL` in `C2` and simplifying the resulting predicate.[6] The first subquery covers the rows in `A` with at least one match in `B`, and the second subquery covers those with no matches.

Feature not supported. The SQL features not supported include `GROUP BY`, `ANY`, `EXISTS`, etc., although they can also be formulated / approximated using basic queries. In the future we plan to leverage other formalisms [14, 18, 19, 66–68, 70] to model complex SQL semantics more precisely.

5.3 Optimizations and SMT Encoding

We end this section with several optimizations for compliance checking and some notes on the SMT encoding.

Strong compliance. We define a stronger notion of compliance, which we found SMT solvers can verify more efficiently.

Definition 5.4. A query Q is *strongly ctx-compliant* to policy $\mathcal{V}$ given trace $\{(Q_i, O_i)\}_{i=1}^{n}$ if for each pair of databases D_1, D_2 that conform to the schema and satisfy:

$$V^{ctx}(D_1) \subseteq V^{ctx}(D_2), \qquad (\forall V \in \mathcal{V}) \tag{4}$$

$$Q_i(D_1) \supseteq O_i, \qquad (\forall 1 \le i \le n) \tag{5}$$

we have $Q(D_1) \subseteq Q(D_2)$.

Theorem 5.5. *If Q is strongly compliant to $\mathcal{V}$ given trace $\mathcal{T}$, then Q is also compliant to $\mathcal{V}$ given $\mathcal{T}$.*

Proof. Let Q be strongly compliant to $\mathcal{V}$ given $\mathcal{T}$. To show that Q is also compliant, let D_1, D_2 be databases that satisfy Equations (1) to (3) from the compliance definition. These imply the strong compliance assumptions (Equations (4) and (5)), and so we have $Q(D_1) \subseteq Q(D_2)$. By symmetry, we also have $Q(D_2) \subseteq Q(D_1)$. Putting the two together, we conclude $Q(D_1) = Q(D_2)$, showing Q to be compliant to $\mathcal{V}$ given $\mathcal{T}$. □

For faster checking, Blockaid verifies strong compliance rather than compliance; by Theorem 5.5, soundness is preserved. However, there are scenarios where a query is compliant but *not* strongly compliant (see our extended paper [73, § C]); such queries will be falsely rejected by Blockaid. This did not pose a problem in practice as we found the two notions to coincide for every query encountered in our evaluation.

Fast accept. Given a view **`SELECT`** `C1, ..., Ck` **`FROM`** `R`, any query that references only columns `R.C1, ..., R.Ck` must be compliant and is accepted without SMT solving.

[5] It suffices to guarantee that Q can be computed from the result of Q'.

[6] As long as `C2` contains no negations, it is safe to treat a `NULL` literal as `FALSE` when propagating through or short-circuiting `AND` and `OR` operators.

Trace pruning. Queries that returns many rows can inflate the trace and slow down the solvers. Fortunately, often times only few of the rows matter to a later query's compliance. We thus adopt a trace-pruning heuristic: when checking a query Q, look for any previous query has returned over ten rows, and keep only those rows that contain the first occurrence of a primary-key value (e.g., user ID) appearing in Q. This heuristic is sound, but may need to be adapted for any application where our premise for pruning does not hold.

SQL types and predicates. To model SQL types, we use SMT's uninterpreted sorts, which we found to yield better performance than theories of integers, strings, etc. We support logical operators `AND` and `OR`, comparison operators <, <=, >, >=, and operators `IN`, `NOT IN`,[7] `IS NULL`, and `IS NOT NULL`. We model < as an uninterpreted relation with a transitivity axiom.

NULLs. We model `NULL` using a two-valued semantics of SQL [31, § 6] by (1) designating a constant in each sort as `NULL`, and (2) taking `NULL` into account when implementing SQL operators. For example, the SQL predicate `x=y` translates into the following SMT formula: $x = y \wedge x \neq null \wedge y \neq null$.

6 Decision Generalization and Caching

While SMT solvers can check a wide range of queries, doing so often takes 100s of milliseconds per query. As a page load can depend on tens of queries, this overhead can add up to *seconds*.

To alleviate this overhead, Blockaid aims to reduce solver calls by caching compliance decisions. Naively, once query Q is deemed compliant given trace $\mathcal{T}$, we could record $(Q, \mathcal{T})$ and allow future occurrences without re-invoking the solvers.

However, this proposal is unlikely to be effective because the number of distinct $(Q, \mathcal{T})$ pairs can be unbounded. For example, an application can issue as many queries of the form `SELECT * FROM Users WHERE UId = ?` as there are users in the system. Therefore, requiring an exact query-trace match for a cache hit would result in a low cache hit rate.

Fortunately, while an application can issue an unbounded number of distinct queries, it only exhibits a finite number of truly different behaviors. For example, the query sequences generated by requests for two different calendar events are likely identical in structure while differing only in parameters (e.g., event ID). If one sequence is compliant, we can *generalize* this knowledge to conclude that the other is also compliant.

This generalization problem is the central challenge we tackle in this section: Given a query's compliance with respect to a trace, how to abstract this knowledge into a *decision template* such that (1) any query (and its trace) that matches this template is compliant, and (2) the template is general enough to produce matches on similar requests. Such a template, once cached, will apply to an entire class of traces and queries.

Decision templates are designed to cache compliant queries only. Our techniques do not extend to non-compliant queries, which are expected to be rare in production as they typically indicate bugs in the application or the policy.

Let us start with an example of a decision template.

6.1 Example

Suppose a user with $UId = 1$ requests Event #42 in the calendar application, resulting in the application issuing a sequence of SQL queries. Consider the third query, shown in Listing 2a. As we explained in Example 4.2, Query #3 is compliant because Query #2 has established that the user attends the event.

Blockaid aims to abstract this query (with trace) into a decision template that applies to another user viewing a different event. Listing 2b shows such a template; the notation says: If each query-output pair above the line has a match in a trace $\mathcal{T}$, then any query of the form below the line is compliant given $\mathcal{T}$. This particular template states: after it is determined that user x attends event y, user x can view event y for any x and y.

Compared with the concrete query and trace, this template (1) omits Query #1, which is immaterial to the compliance decision; and (2) replaces the concrete values with parameters. Occurrences of `?0` here constrain the event ID fetched by the query to equal the previously checked event ID. We use `*` to denote a fresh parameter, i.e., any arbitrary value is allowed.

We now dive into how Blockaid extracts such a decision template from a concrete query and trace. But before we do so, let us first define what a decision template is, what it means for a template to have a match, and what makes a "good" template.

6.2 Definitions and Goals

For convenience, from now on we will denote a trace as a set of query-*tuple* pairs $\{(Q_i, t_i)\}_{i=1}^{n}$, where each t_i is *one* of the rows returned by Q_i. A query that returns multiple rows is represented as multiple such pairs. This change of notation is permissible because under strong compliance (Definition 5.4), we no longer take into account the *absence* of a returned row.

Definition 6.1. We say a trace $\mathcal{T} = \{(Q_i, t_i)\}_{i=1}^{n}$ is *feasible* if there exists a database D such that $t_i \in Q_i(D)$ for all $1 \leq i \leq n$.

Definition 6.2. A *decision template* $\mathcal{D}[\mathbf{x}, \mathbf{c}]$, where $\mathbf{c}$ denotes variables from the request context and $\mathbf{x}$ a sequence of variables disjoint from $\mathbf{c}$, is a triple $(Q_\mathcal{D}, \mathcal{T}_\mathcal{D}, \Phi_\mathcal{D})$ where:

- $Q_\mathcal{D}$ is the *parameterized query*, whose definition can refer to variables from $\mathbf{x} \cup \mathbf{c}$;
- $\mathcal{T}_\mathcal{D}$ is the *parameterized trace*, whose queries and tuples can refer to variables from $\mathbf{x} \cup \mathbf{c}$; and
- $\Phi_\mathcal{D}$, the *condition*, is a predicate over $\mathbf{x} \cup \mathbf{c}$.

We will often denote a template simply by $\mathcal{D}$ if the variables are either unimportant or clear from the context.

As we later explain, $\Phi_\mathcal{D}$ represents any extra constraints that a template imposes on its variables (e.g., `?0` < `?1`).

Definition 6.3. A *valuation* ν over a collection of variables $\mathbf{y}$ is a mapping from $\mathbf{y}$ to constants (including `NULL`), extended to objects that contain variables in $\mathbf{y}$. For example, given a parameterized query Q, $\nu(Q)$ denotes Q with each occurrence of variable $y \in \mathbf{y}$ substituted with $\nu(y)$.

[7] We only support `IN` and `NOT IN` with a list of values, not with a subquery.

Listing 2: *An example query with trace from the calendar application and a decision template generated from it.*

(a) *Example query with trace ($UId = 1$).*

```
1. SELECT * FROM Users WHERE UId = 1
   ↪ (UId=1, Name="John Doe")
2. SELECT * FROM Attendances
   WHERE UId = 1 AND EId = 42
   ↪ (UId=1, EId=42, ConfirmedAt="05/04 1pm")
---
3. SELECT * FROM Events WHERE EId = 42
```

(b) *The decision template generated by Blockaid.*

```
1. SELECT * FROM Attendances
   WHERE UId = ?MyUId AND EId = ?0
   ↪ (UId = ?MyUId, EId = ?0, ConfirmedAt = *)
---
✓ SELECT * FROM Events WHERE EId = ?0
```

Definition 6.4. Let $\mathcal{D}[\mathbf{x},\mathbf{c}] = (Q_\mathcal{D}, \mathcal{T}_\mathcal{D}, \Phi_\mathcal{D})$ be a decision template, *ctx* be a request context, $\mathcal{T}$ be a trace, and Q be a query. We say that $\mathcal{D}$ *matches* $(Q,\mathcal{T})$ *under ctx* if there exists a valuation ν over $\mathbf{x} \cup \mathbf{c}$ such that:

- $\nu(\mathbf{c}) = ctx$,
- $\nu(Q_\mathcal{D}) = Q$,
- $(\nu(Q_j), \nu(t_j)) \in \mathcal{T}$ for all $(Q_j, t_j) \in \mathcal{T}_\mathcal{D}$, and
- $\nu(\Phi_\mathcal{D})$ holds.

Example 6.5. Listing 2b can be seen as a stylized rendition of a decision template $\mathcal{D}[\mathbf{x},\mathbf{c}]$ where $\mathbf{x} = (x_0, x_1)$—x_0 denoting `?0` and x_1 denoting the occurrence of `*`—and $\mathbf{c} = (MyUId)$; $Q_\mathcal{D}$ and $\mathcal{T}_\mathcal{D}$ are as shown below and above the line; and $\Phi_\mathcal{D}$ is the constant $\top$, meaning the template imposes no additional constraints on the variables.[8] Under the request context $MyUId = 1$, this template matches the query and trace in Listing 2a via the valuation $\{x_0 \mapsto 42, x_1 \mapsto$ `"05/04 1pm"`$, MyUId \mapsto 1\}$.

We are interested only in templates that imply compliance.

Definition 6.6. A decision template $\mathcal{D}$ is *sound* with respect to a policy $\mathcal{V}$ if for every request context *ctx*, whenever $\mathcal{D}$ matches $(Q,\mathcal{T})$ under *ctx*, Q is strongly *ctx*-compliant to $\mathcal{V}$ given $\mathcal{T}$.

Blockaid can verify that a template is sound via the following theorem derived from strong compliance (Definition 5.4):

Theorem 6.7. *A decision template $\mathcal{D}[\mathbf{x},\mathbf{c}] = (Q_\mathcal{D}, \mathcal{T}_\mathcal{D}, \Phi_\mathcal{D})$ is sound with respect to a policy $\mathcal{V}$ if and only if:*

$$\forall \mathbf{x}, \mathbf{c}, D_1, D_2. \quad \left.\begin{array}{r} \Phi_\mathcal{D} \\ \forall V \in \mathcal{V}. V(D_1) \subseteq V(D_2) \\ \forall (Q_i, t_i) \in \mathcal{T}_\mathcal{D}. t_i \in Q_i(D_1) \end{array}\right\} \implies Q_\mathcal{D}(D_1) \subseteq Q_\mathcal{D}(D_2).$$

For a compliant query Q (with trace $\mathcal{T}$) that misses the cache, there often exist many sound templates that match $(Q,\mathcal{T})$. But all such templates are not equal—we prefer the more *general* ones, those that match a wider range of *other* queries and traces.

Definition 6.8. A template $\mathcal{D}_1$ is *at least as general as* a template D_2 if for every query Q and feasible trace $\mathcal{T}$, if $\mathcal{D}_2$ matches $(Q,\mathcal{T})$, $\mathcal{D}_1$ also matches $(Q,\mathcal{T})$.

[8] Technically, this template requires $MyUId \neq$ `NULL` $\wedge\, x_0 \neq$ `NULL`. We omitted this condition in Listing 2b because we assume the user ID parameter and the *Attendances* table's *EId* column are both non-NULL.

Thus, Blockaid aims to generate a decision template that (1) is sound, (2) matches $(Q,\mathcal{T})$, and (3) is general enough for practical purposes. We now explain how this is achieved.

6.3 Generating Decision Templates

Blockaid starts from the trivial template $D_0 = (Q, \mathcal{T}, \top)$, which is sound but not general, and generalizes it in two steps:

1. Minimize the trace $\mathcal{T}$ to retain only those (Q_i, t_i) pairs that are required for Q's compliance (§6.3.1).
2. Replace each constant in the trace and query with a fresh variable, and then generate a weak condition Φ over the variables that guarantees compliance (§6.3.3).

6.3.1 Step One: Trace Minimization

Blockaid begins by finding a minimal sub-trace of $\mathcal{T}$ that preserves compliance. It removes each $(Q_i, t_i) \in \mathcal{T}$ and, if Q is no longer compliant, adds the element back. For example, for Listing 2a this step removes Query #1. Denote the resulting minimal trace by $\mathcal{T}_\text{min}$ and let decision template $\mathcal{D}_1 = (Q, \mathcal{T}_\text{min}, \top)$.

Proposition 6.9. *$\mathcal{D}_1$ is sound, matches $(Q,\mathcal{T})$, and is at least as general as $\mathcal{D}_0$.*

As an optimization, Blockaid starts the minimization from the sub-trace that the solver has actually used to prove compliance. It extracts this information from a solver-generated *unsat core* [8, § 11.8]—a subset of clauses in the formula that remains unsatisfiable even with all other clauses removed. If we attach *labels* to the clauses we care about, a solver will identify all labels in the unsat core when it proves the formula unsatisfiable.

To get an unsat core, Blockaid uses the following formula:

$$[LQ_i] \quad \begin{array}{rr} V^{ctx}(D_1) \subseteq V^{ctx}(D_2), & (\forall V \in \mathcal{V}) \\ t_i \in Q_i(D_1), & (\forall (Q_i, t_i) \in \mathcal{T}) \\ Q(D_1) \not\subseteq Q(D_2), & \end{array}$$

where the clause asserting the i^{th} trace entry is labeled LQ_i. If Q is compliant, the solver returns as the unsat core a set S of labels. Blockaid ignores any $(Q_i, t_i) \in \mathcal{T}$ for which $LQ_i \notin S$.

6.3.2 Interlude: Model Finding for Satisfiable Formulas

A common operation in template generation is to remove parts of a formula and re-check satisfiability. A complication arises

when the formula turns satisfiable—while solvers are adept at proving unsatisfiability, they often fail on satisfiable formulas.[9]

To solve these formulas faster, we observe that they are typically satisfied by databases with small tables. We thus construct SMT formulas to directly seek such "small models" by representing each table not as an uninterpreted relation, but as a conditional table [35] whose size is bounded by a small constant.

A conditional table generalizes a regular table by (1) allowing variables in its entries, and (2) associating with each row with a *condition*, i.e., a Boolean predicate for whether the row exists. For example, a *Users* table with a bound of 2 appears as:

UId	*Name*	Exists?
$x_{u,1}$	$x_{n,1}$	b_1
$x_{u,2}$	$x_{n,2}$	b_2

where each entry and condition is a fresh variable, signifying that the table is not constrained in any way other than its size.

Queries on condition tables are evaluated via an extension of the relational algebra operators [35, § 7]. This allows queries to be encoded into SMT without using quantifiers or using relation symbols for tables.[10] For example, the query `SELECT Name FROM Users WHERE UId = 5` can be written as:

$$\mathsf{Q}(\mathsf{x}_n) := \bigvee_{i=1}^{2} (x_{u,i} = 5 \wedge x_{n,i} = \mathsf{x}_n \wedge b_i)\,.$$

We found that such formulas could be solved quickly by Z3.

After Blockaid generates an unsat core as described in §6.3.1, it switches to using bounded formulas (i.e., ones that use conditional tables instead of uninterpreted relations) for the remainder of template generation. Blockaid sets a table's bound to one plus the number of rows required to produce the sub-trace induced by the unsat core;[11] it relies on the solvers to produce small unsat cores to keep formula sizes manageable.

Care must be taken because using bounded formulas breaks soundness—a query compliant on small tables might not be on larger ones. Therefore, after a decision template is produced Blockaid verifies its soundness on the unbounded formula, and if this fails, increments the table bounds and retry.

6.3.3 Step Two: Find Value Constraints

Taking the template $\mathcal{D}_1 = (Q, \mathcal{T}_{\min}, \top)$ from Step 1, Blockaid generalizes it further by abstracting away the constants. To do so, Blockaid *parameterizes* $\mathcal{T}_{\min}$ and Q by replacing each occurrence of a constant with a fresh variable. We use a superscript "p" to denote the parameterized version of a query, tuple, or trace. Listing 3a shows $\mathcal{T}^{\mathrm{p}}_{\min}$ and Q^{p} from our example. As an optimization, Blockaid assigns the same variable (e.g., `x0`) to locations that are guaranteed by SQL semantics to be equal.

[9] For example, finite model finders in CVC4 [57] and Vampire [56] often time out or run out of memory on tables with only tens of columns.

[10] To avoid using quantifiers in these formulas, we drop the transitivity axiom for the uninterpreted less-than relation (§5.3).

[11] If the bounds are too small for a database to produce the trace, the resulting formula will be unsatisfiable regardless of compliance.

Listing 3: *Parameterization and candidate atoms for Listing 2a.*

(a) *Parameterized trace $\mathcal{T}^{p}_{min}$ and query q^{p}.*

```
2. SELECT * FROM Attendances
   WHERE UId = x0 AND EId = x1
   ↪ (UId = x0, EId = x1, ConfirmedAt = x2)
3. SELECT * FROM Events WHERE EId = x3
```

(b) *Candidate atoms (with symmetric duplicates removed).*

Form `x = v`:
- `MyUId = 1`
- `x0 = 1`
- `x1 = 42`
- `x2 = "05/04 1pm"`
- `x3 = 42`

Form `x = x'`:
- `MyUId = x0`
- `x1 = x3`

Form `x < x'`:
- `MyUId < x1`
- `MyUId < x3`
- `x0 < x1`
- `x0 < x3`

Blockaid must now generate a condition Φ such that the resulting template $\mathcal{D}_2 = (Q^{\mathrm{p}}, \mathcal{T}^{\mathrm{p}}_{\min}, \Phi)$ meets our goals. It picks as Φ a conjunction of atoms from a set of *candidate atoms*. Let $\mathbf{x}$ denote all variables generated from parameterization, and let ν map $\mathbf{x}$ to the replaced constants and $\mathbf{c}$ to the current context *ctx*.

Definition 6.10. The set of *candidate atoms* is defined as:

$$C = \bigcup \begin{cases} \{\texttt{x = v} & \mid x \in \mathbf{x} \cup \mathbf{c}, v = \nu(x) \neq \texttt{NULL}\} \\ \{\texttt{x IS NULL} & \mid x \in \mathbf{x} \cup \mathbf{c}, \nu(x) = \texttt{NULL}\} \\ \{\texttt{x = x'} & \mid x, x' \in \mathbf{x} \cup \mathbf{c}, \nu(x) = \nu(x') \neq \texttt{NULL}\} \\ \{\texttt{x < x'} & \mid x, x' \in \mathbf{x} \cup \mathbf{c}, \nu(x) < \nu(x')\} \end{cases}.$$

(We write atoms in `monospace font` to distinguish them from mathematical expressions. Following SQL, the "=" in an atom implies that both sides are non-`NULL`.)

Note that all candidate atoms hold on Q and $\mathcal{T}_{\min}$. Blockaid now selects a subset that not only guarantees compliance, but also imposes relatively few restrictions on the variables.

Definition 6.11. With respect to Q^{p} and $\mathcal{T}^{\mathrm{p}}_{\min}$, a subset of atoms $C_0 \subseteq C$ is *sound* if the decision template $(Q^{\mathrm{p}}, \mathcal{T}^{\mathrm{p}}_{\min}, \bigwedge C_0)$ is sound. ($\bigwedge C_0$ denotes the conjunction of atoms in C_0.)

Definition 6.12. Let $C_1, C_2 \subseteq C$. We say that C_2 is *at least as weak as* C_1 (denoted $C_1 \preceq C_2$) if $\bigwedge C_1 \implies \bigwedge C_2$, and that C_2 is *weaker* than C_1 if $C_1 \preceq C_2$ but $C_2 \not\preceq C_1$.

Example 6.13. Listing 3b shows all the candidate atoms from Listing 3a (after omitting symmetric ones in the `x = x'` group). Consider the following two subsets of atoms:

$$C_1 = \{\texttt{MyUId = x0},\ \texttt{x1 = 42},\ \texttt{x3 = 42}\},$$
$$C_2 = \{\texttt{MyUId = x0},\ \texttt{x1 = x3}\}.$$

While both are sound, C_2 is preferred over C_1 as it is weaker and thus applies in more scenarios. In fact, C_2 is *maximally* weak: there exists no subset that is both sound and weaker than C_2.

Ideally, Blockaid would produce a maximally weak sound subset of C to use as the template condition, but finding one can be expensive. It thus settles for finding a subset that is weak enough for practical generalization. It does so in three steps.

First, as a starting point, Blockaid generates a minimal unsat core of the formula:

$$\begin{aligned} & V^{ctx}(D_1) \subseteq V^{ctx}(D_2), && (\forall V \in \mathcal{V}) \\ & t_i^{\mathrm{p}} \in Q_i^{\mathrm{p}}(D_1), && (\forall (t_i^{\mathrm{p}}, Q_i^{\mathrm{p}}) \in \mathcal{T}_{\min}^{\mathrm{p}}) \\ [LC_i] \quad & c_i, && (\forall c_i \in C) \\ & Q^{\mathrm{p}}(D_1) \not\subseteq Q^{\mathrm{p}}(D_2). \end{aligned}$$

Let C_{core} denote the atoms whose label appears in the unsat core. For example, $C_{\text{core}} = \{\texttt{MyUId = x0}, \texttt{x1 = 42}, \texttt{x3 = 42}\}$.

Second, it augments C_{core} with other atoms that are implied by it: $C_{\text{aug}} = \{c \in C \mid \bigwedge C_{\text{core}} \implies c\}$. In our example,

$$\begin{aligned} C_{\text{aug}} &= C_{\text{core}} \cup \{\texttt{x1 = x3}\} \\ &= \{\texttt{MyUId = x0}, \quad \texttt{x1 = 42}, \quad \texttt{x3 = 42}, \quad \texttt{x1 = x3}\}. \end{aligned}$$

C_{aug} enjoys a closure property: if $C_0 \subseteq C_{\text{aug}}$ and $C_0 \preceq C_1$, then $C_1 \subseteq C_{\text{aug}}$. In particular, C_{aug} contains a maximally weak sound subset of C. Thus, Blockaid focuses its search within C_{aug}.

Finally, as a proxy for weakness, Blockaid finds a *smallest* sound subset of C_{aug}, denoted C_{small}, breaking ties arbitrarily. It does so using the MARCO algorithm [43, 44, 55] for minimal unsatisfiable subset enumeration, modified to enumerate from small to large and to stop after finding the first sound subset. In our example, the algorithm returns $C_{\text{small}} = \{\texttt{MyUId=x0}, \texttt{x1=x3}\}$ of cardinality two, which is also a maximally weak subset (even though this might not be the case in general).[12] Nevertheless, searching for a smallest sound subset has produced templates that generalize well in practice.

At the end, Blockaid produces the decision template:

$$\mathcal{D}_2[\mathbf{x}, \mathbf{c}] = \left(Q^{\mathrm{p}}, \mathcal{T}_{\min}^{\mathrm{p}}, \bigwedge C_{\text{small}}\right).$$

Proposition 6.14. *$\mathcal{D}_2$ is sound, matches $(Q, \mathcal{T})$, and is at least as general as $\mathcal{D}_1$.*

As an optimization, whenever $\bigwedge C_{\text{small}} \implies x = y$ for $x, y \in \mathbf{x} \cup \mathbf{c}$, Blockaid replaces x with y in the template. This is how, e.g., in Listing 2b `?0` appears in both the trace and the query.

6.3.4 Optimizations

We implement two optimizations that improve the performance of template generation and the generality of templates.

Omit irrelevant tables. Given trace $\mathcal{T}$ and query Q, we call a table *relevant* if (1) it appears in $\mathcal{T}$ or Q, or (2) the table appears on the right-hand side of a database constraint of the form $Q_1 \subseteq Q_2$, given that a relevant table appears on the left.[13] Blockaid sets the size bounds of irrelevant tables to zero, reducing formula size while preserving compliance.

Split `IN`. A query Q that contains "c `IN` $(x_1, x_2, \ldots, x_n)$" often produces a template with a long trace. If Q is a basic query that does not contain the `NOT` operator, it can be split into $q_1, \ldots, q_n$ where q_i denotes Q with the `IN`-construct substituted with $c = x_i$, such that $Q \equiv q_1 \cup \ldots \cup q_n$. If $q_1, \ldots, q_n$ are all compliant then so is Q, and so Blockaid checks the subqueries instead. This is usually fast because $q_2, \ldots, q_n$ typically match the decision template generated from q_1. If any q_i is not compliant, Blockaid reverts to checking Q as a whole.

This optimization also improves generalization. Suppose Q' has structure identical to Q but a different number of `IN` operands. It would not match a template generated from Q, but its split subqueries q_i' could match the template from q_1.

6.4 Decision Cache and Template Matching

Blockaid stores decision templates in its *decision cache*, indexing them by their parameterized query using a hash map. When checking a query Q, Blockaid lists all templates whose parameterized query matches Q; for each such template, it uses recursive backtracking (with pruning optimizations) to search for a valuation that results in a match. This simple method proves efficient in practice as the templates tend to be small.

7 Implementation

We implemented Blockaid as a Java Database Connectivity (JDBC) driver that interposes on an underlying connection. It thus supports only applications on the JVM and runs within the web server, although our design allows it to reside elsewhere (e.g., in the database). The JDBC driver accepts custom commands that (1) set the request context, (2) clear the context and the trace, and (3) check an application cache read.

Blockaid parses SQL using Apache Calcite [9] and caches parser outputs. To check compliance, it uses Z3's Java binding [23] to generate formulas in SMT-LIB 2 format [7] and invokes an ensemble of solvers in parallel. Our ensemble consists of Z3 [24] (v4.8.12) and CVC5 [6] (v0.3) using default configurations, and Vampire [40] (v4.6.1) using six configurations from its CASC portfolio.[14] The ensemble is killed as soon as any solver finishes. If a query is not compliant, or all solvers time out after 5 s, Blockaid throws a Java `SQLException`.

To generate decision templates, Blockaid uses the same ensemble to produce the initial unsat core (§6.3.1), but kills the ensemble only when a solver returns a small core of up to 3 labels (subject to timeout). It uses only Z3 on bounded formulas.

Our prototype does not verify that queries return no duplicate rows and does not look at any `ORDER BY` columns. We manually ensured that queries in our evaluation return no duplicates and do not reveal inaccessible information through `ORDER BY`.

[12] For example, $\{\texttt{x < y}, \texttt{x < z}\}$ is strictly weaker than $\{\texttt{x < y}, \texttt{y < z}\}$ even though the two sets have the same cardinality.

[13] Every constraint encountered in our evaluation can be written in the form $Q_1 \subseteq Q_2$, including primary-key, foreign-key, and integrity constraints.

[14] https://github.com/vprover/vampire/blob/master/CASC/Schedules.cpp.

Table 1: Summary of schemas, policies, and code changes.

	diaspora*	Spree	Autolab
Schema & Policy			
# Tables modeled	35 / 52	46 / 93	17 / 28
# Constraints	108	122	51
# Policy views	108	84	57
# Cache key patterns	0	11	3
Code Changes (LoC)			
Boilerplate	12	17	12
Fetch less data	6	26	38
SQL feature	1	3	5
Parameterize queries	0	18	32
File system checking	0	0	9
Total	19	64	96

8 Evaluation

We use Blockaid to enforce data-access policies on three existing open-source web applications written in Ruby on Rails:

- **diaspora*** [25]: a social network with 850 k users.
- **Spree** [63]: an e-commerce app used by 50+ businesses.
- **Autolab** [5]: a course management app used at 20 schools.

For each application, we devised a data-access policy, modified its code to work with Blockaid, and measured its performance.

In summary: Blockaid imposes overheads of 2 %–12 % to median page load time when compliance decisions are cached; the decision templates produced by Blockaid generalize to other entities (users, etc.); and no query was falsely rejected in our benchmark. Instructions for reproducing our experiments can be found in Appendix A.

8.1 Constraints, Policies, and Annotations

Table 1 summarizes the constraints and policies for database tables queried in our benchmark, including any necessary application-level constraints (e.g., a reshared post is always public in diaspora*). Spree and Autolab use the Rails cache, and we annotate their cache key patterns with queries (§3.2).

Once a policy is given, transcribing it into views was straightforward. The more arduous task lied in divining the intended policy for an application, by studying its source code and interacting with it on sample data. This effort was complicated by edge cases in policies—e.g., a Spree item at an inactive location is inaccessible *except* when filtering for backorderable variants. Such edge cases had to be covered using additional views.

To give a sense of the porting effort, writing the Spree policy took one of us roughly a month. However, this process would be easier for the developer of a new application, who has a good sense of what policies are suitable and can create policies while building the application, amortizing the effort over time.

When writing the Autolab policy, we uncovered two access-check bugs in the application: (1) a persistent announcement (one shown on all pages of a course) is displayed regardless of whether it is active on the current date, and (2) an unreleased handout is hidden on its course page but can be downloaded from its assignment page. This experience corroborates the difficulty of making every access check airtight, especially for code bases that enjoy fewer maintenance resources.

8.2 Code Modifications

Our changes to application code fall into five categories:

1. **Boilerplate**: We add code that sends the request context to Blockaid at request start and clears the trace at request end.
2. **Fetch less data**: We modify code to not fetch potentially sensitive data unless it will be revealed to the user; some of these changes use the `lazy_column` gem [45].
3. **SQL features**: We modify some queries to avoid SQL features not supported by Blockaid (e.g., general left joins) without altering application behavior.
4. **Parameterize queries**: We make some queries parameterized so that Blockaid can effectively cache their parsing results. Most changes are mechanical rewrites of queries with comparisons, as idiomatic ways of writing comparisons [54] cause query parameters to be filled within Rails.
5. **File system checking**: Autolab uses files to store submissions; the file name are always accessible but the content is inaccessible during an exam. We modify it to store the submission content under a randomly generated file name and restrict access to the file name in the database (§3.2).

The code changes are summarized also in Table 1, which omits configuration changes, adaptations for JRuby, and experiment code. The changes range from 19 to 96 lines of code.

8.3 Experiment Setup and Benchmark

We deploy each application on an Amazon EC2 c4.8xlarge instance running Ubuntu 18.04. Because our prototype only supports JVM applications (§7), we run the applications using JRuby [36] (v9.3.0.0), a Ruby implementation atop the JVM (we use OpenJDK 17). In Rails's database configuration, we turn on `prepared_statements` so that Rails issues parameterized queries in the common case.[15] The applications run atop the Puma web server over HTTPS behind NGINX (which serves static files directly), and stores data in MySQL (and, if applicable, Redis) on the same instance. To reduce variability, all measurements are taken from a client on the same instance.

For each application, we picked five page loads that exercise various behaviors (Table 2). Each page load can fetch multiple URLs, some common among many pages (e.g., D9, which is the notifications URL). All queries issued are compliant, and all experiments are performed with the Rails cache populated.

8.4 Page Load Times

We start by measuring page load times (PLTs) using a headless Chrome browser (v96) driven by Selenium [62]. PLTs are reported as the time elapsed between `navigationStart` and `loadEventEnd` as defined by the `PerformanceTiming` interface [71]. The one exception is the Autolab "Submission" page,

[15] In case a Rails query is not fully parameterized (e.g., due to the use of raw SQL), it gets parameterized by Blockaid as described in §6.3.3.

Table 2: *Application benchmark. For a page we list the page URL followed by other URLs fetched (URLs for assets are excluded). When compliance decisions are cached, Blockaid incurs up to* 12 % *overhead to the median PLT over the modified applications.*

			Page Load Time (median / P95; default unit: ms)			
	URLs	**Description**	Original	Modified	Cached	No cache
diaspora*						
Simple post	D1, D2, D9	View a simple post shared with the user.	169 / 173	169 / 175	174 / 179	2.5 s / 2.6 s
Complex post	D3, D4, D9	View a public post with 30 votes and comments.	171 / 178	171 / 178	176 / 183	2.6 s / 2.7 s
Prohibited post	D5	Attempt to view an unauthorized post.	32 / 34	32 / 34	33 / 35	262 / 285
Conversation	D6, D9	View a conversation (5 messages).	253 / 258	255 / 262	260 / 267	2.1 s / 2.2 s
Profile	D7, D8, D9	View someone's profile (basic info and 3 posts).	142 / 148	145 / 152	150 / 156	1.3 s / 1.4 s
Spree						
Account	S1, S6–S8	View the user's account information.	74 / 80	76 / 83	78 / 84	588 / 611
Available item	S2, S6–S8	View a product for sale.	122 / 133	115 / 167	122 / 173	4.4 s / 4.4 s
Unavailable item	S3	Attempt to view a product no longer for sale.	20 / 22	21 / 23	22 / 24	350 / 371
Cart	S4, S6–S8	View the current shopping cart (3 items).	116 / 131	118 / 132	124 / 137	7.6 s / 7.7 s
Order	S5, S6–S8	View a summary and status of a previous order.	160 / 170	164 / 174	173 / 182	39 s / 39 s
Autolab						
Homepage	A1	View a summary of 3 courses enrolled.	56 / 61	59 / 64	65 / 70	1.4 s / 1.6 s
Course	A2, A3	View summary of one course (15 assignments).	84 / 96	87 / 101	97 / 116	3.9 s / 4.1 s
Assignment	A4	View a quiz (incl. 3 submissions and grades).	97 / 110	103 / 118	115 / 138	3.5 s / 3.6 s
Submission	A5	Download a previous homework submission.	22 / 26	26 / 31	27 / 33	1.1 s / 1.2 s
Gradesheet	A6	Instructor views grades for 51 enrollees.	456 / 474	474 / 493	504 / 530	72 s / 73 s

a file download, for which we report Chrome's download time instead. Since the client is on the same VM as the server, these experiments reflect the best-case PLT, as clients outside the instance / cloud are likely to experience higher network latency.

We report PLTs under four settings: *original* (unmodified application), *modified* (modified à la §8.2), *cached* (modified application under Blockaid with every query hitting the decision cache), and *no cache* (decision caching disabled). For the first three, we perform 3000 warmup loads before measuring the PLT of another 3000 loads. For *no cache*, where each run takes longer, we use 100 warmup loads and 100 measurement loads.

Table 2 shows that when compliance decisions are cached, Blockaid incurs up to 12 % overhead to median PLT over the modified application (and up to 17 % overhead to P95). With caching disabled, Blockaid incurs up to 236× higher median PLT. Compared with the original applications, the modified versions result in up to 6 % overhead to median PLT for all pages but Autolab's "Submissions", which suffers a 19 % overhead. (The P95 overhead is up to 7 % for all but two pages with up to 26 % overhead.) We will comment on these overheads in the next subsection, where we break down the pages into URLs.

8.5 Fetch Latency

To better understand page load performance, we separate out the individual URLs fetched by each page (Table 2), omitting URLs for assets, and measure the latency of fetching each URL (not including rendering time). The median latencies are shown in Figure 2. In addition to the four settings from §8.4, it includes performance under a "cold cache", where the decision cache is enabled but cleared at the start of each load (100 warmup runs followed by 100 measurements). When all compliance decisions are cached, Blockaid incurs up to 10 % of overhead (median 7 %) over "modified". In contrast, it incurs 7×–422× overhead on a cold decision cache, and 7×–310× overhead if the decision cache is disabled altogether.

For most URLs, "cold cache" is slower than "no cache" due to the extra template-generation step. Two exceptions are D4 and A6, where many structurally identical queries are issued, and so the performance gain from cache hits *within each URL* offsets the performance hit from template generation.

Compared to the original, the modified diaspora* and Spree are up to 5 % slower (median 2 %), but Autolab is up to 21 % slower (median 8 %). Autolab routinely reveals partial data on objects that are not fully accessible. For example, a user can distinguish among the cases where (1) a course doesn't exist, (2) a course exists but the user is not enrolled, and (3) the user is enrolled but the course is disabled. The original Autolab fetches the course in one SQL query but we had to split it into multiple—checking whether the course exists, whether it is disabled, etc.—and return an error immediately if one of these checks fails.

In one instance (S2), the modified version is 11 % faster than the original because we were able to remove queries for potentially inaccessible data that is never used in rendering the URL.

8.6 Solver Comparison

When a query arrives, Blockaid invokes an ensemble of solvers to check compliance when decision caching is disabled, and to generate a decision template on a cache miss when caching is enabled. The *winner*, in the no-cache case, is the first solver to return a decision; and in the cache-miss case, the first to return a

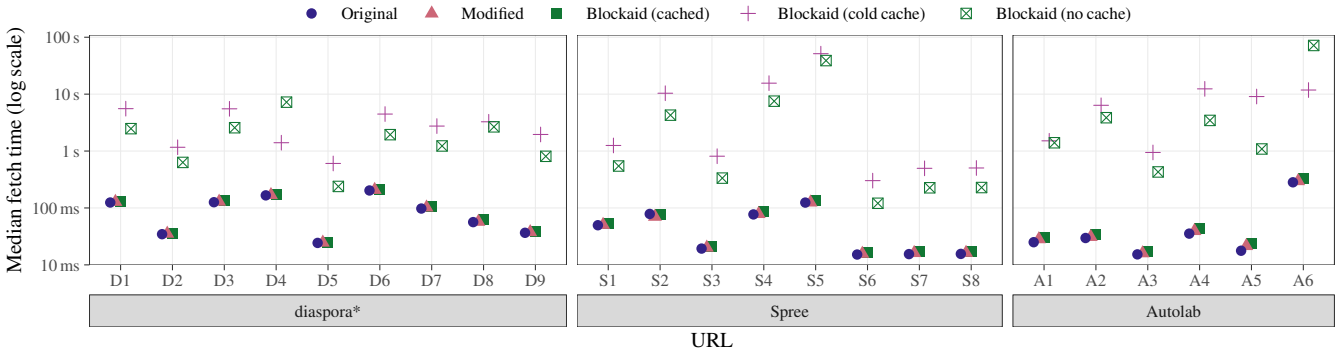


Figure 2: *URL fetch latency (median). With all compliance decisions cached, Blockaid incurs up to* 10 *% overhead over "modified".*

Listing 4: *Two (abridged) decision templates generated for the same parameterized query from Spree.* `Token` *is a Spree request context parameter identifying the current (possibly guest) user, and* `NOW` *is a built-in parameter storing the current time.*

(a) *This template doesn't fully generalize.*

```
SELECT * FROM products WHERE id IN (*, *, *)
↪ (id = ?1, available_on < ?NOW,
    discontinue_on IS NULL, deleted_at IS NULL, *)

SELECT * FROM variants WHERE id IN (*, *, *)
↪ (id = ?2, deleted_at IS NULL,
    discontinue_on IS NULL, product_id = ?1, *)
```

```
SELECT a.* FROM assets a
JOIN variants mv ON a.viewable_id = mv.id
JOIN variants ov ON mv.product_id = ov.product_id
WHERE mv.is_master AND mv.deleted_at IS NULL
  AND a.viewable_type = 'Variant' AND ov.id = ?2
```

(b) *This template does fully generalize.*

```
SELECT * FROM orders WHERE ...
↪ (id = ?0, token = ?Token, *)

SELECT * FROM line_items WHERE order_id = ?0
↪ (variant_id = ?1, *)
```

```
SELECT a.* FROM assets a
JOIN variants mv ON a.viewable_id = mv.id
JOIN variants ov ON mv.product_id = ov.product_id
WHERE mv.is_master AND mv.deleted_at IS NULL
  AND a.viewable_type = 'Variant' AND ov.id = ?1
```

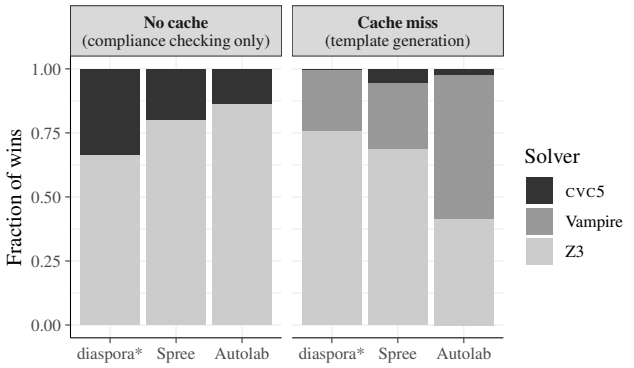


Figure 3: *Fraction of wins by each solver. "Vampire" covers a portfolio of six configurations (§7).*

small enough unsat core (§7), assuming the query is compliant.

Figure 3 shows, in the fetch latency experiments (§8.5), the fraction of wins by each solver in the two cases. In the no-cache case, the wins are dominated by Z3 followed by CVC5, with none for Vampire. In the cache-miss case, however, Vampire wins a significant portion of the time. This is because Z3 and CVC5 often finish quickly but with large unsat cores, causing Blockaid to wait till Vampire produces a smaller core.

8.7 Template Generalization

We found that the generated decision templates typically generalize to similar requests. The rest generalize in more restricted scenarios, and none is tied to a particular user ID, post ID, etc.

To illustrate how Blockaid might produce a template that fails to generalize fully, consider a query from Spree's cache key annotations (Listing 4). This query fetches assets for product variants in the user's order. (Here, the asset of a variant belongs to its product's "master variant".) Listing 4a shows a template that fails to generalize fully, for three reasons.

First, due to the queries with the `IN` operator in its premise (above the horizontal line), this template applies only when an order has exactly three variants. The `IN`-splitting optimization from §6.3.4 only applies to the query being checked, and we plan to handle such queries in the premise in future work.

Second, this template constrains the variant to be "not discontinued", which is defined as `discontinue_on IS NULL` or `discontinue_on >= NOW`. But because disjunctions are not supported in decision templates, Blockaid picked only the condition that matches the current variant (`IS NULL`).

Third, in this example there are multiple justifications for

this query's compliance, and Blockaid happened to pick one that does not always hold in a similar request. The policy states that a variant's asset can be viewed if it is not discontinued, or if it is part of the user's order.[16] This particular variant in the user's order happens to not be discontinued, and the template captures the former justification for viewing the asset. However, it does not apply to variants in the order that *are* discontinued; indeed, for such variants, Blockaid produces the template in Listing 4b, which generalizes fully. We could address this issue by finding multiple decision templates for every query.

Incidentally, inspecting decision templates has helped us expose overly permissive policies. When writing the Autolab policy, we missed a join condition in a view, a mistake that became apparent when Blockaid generated a template stating that an instructor for one course can view assignments for *all* courses. Although manual inspection of templates is not required for using Blockaid, doing so can help debug overly broad policies, whose undesirable consequences are often exposed by the general decision templates produced by Blockaid.

9 Additional Issues

Comparison to row- and cell-level policy. Several commercial databases (such as SQL Server [49] and Oracle [52]) implement *row- and/or cell-level* data-access policies, which specify accessible information at the granularity of rows or cells.

Such policies are less expressive than the view-based ones supported by Blockaid. For example, suppose we wish to allow each user to view everyone's timetables (i.e., the start and end times of the events they attend). Querying someone's timetable requires joining the *Events* and *Attendances* tables on the *EId* column, which must then be treated as visible by a cell-level policy. But this inevitably reveals meeting attendee information as well. Instead, we can implement this policy using a view:

```
SELECT UId, StartTime, EndTime
FROM   Events e
JOIN   Attendances a ON e.EId = a.EId
```

which lists the times of events attended without revealing *EId*.

False rejections. Even though false rejections of compliant queries never occurred in our evaluation, they remain a possibility for several reasons, including: (1) approximate rewriting into basic queries, which is incomplete; (2) our use of strong compliance; and (3) solver timeouts. Developers can reduce the chance of false rejections by running an application's end-to-end test suite under Blockaid before deployment, and manually examining any rejected query to determine whether it is due to a false positive, a bug in the code, or a misspecified policy.

Off-path deployment. If an operator is especially worried about false rejections affecting a website's availability, we can modify Blockaid to log potential violations instead of blocking any queries. We can even move Blockaid off-path by having the application stream its queries to Blockaid to be checked asynchronously, further reducing its performance impact.

[16] This is to allow users to view past purchases that are since discontinued.

What if Blockaid could issue its own queries? Suppose Blockaid can issue extra queries—but only ones answerable using the views, lest the decision itself reveal sensitive data—when checking compliance. Blockaid can now safely allow more queries from the application. For example, faced with the formerly non-compliant single query from Example 4.3:

```
SELECT Title FROM Events WHERE EId = 5
```

Blockaid can now *ask* whether the user attends Event #5 and if so, allow the query. In fact, under this setup the "necessary-and-sufficient" condition for application noninterference (in the sense of §4.3) becomes instance-based determinacy [39,58,74], a criterion less stringent than trace determinacy.

We decided against this design alternative for two reasons. First, it seems nontrivial to check instance-based determinacy efficiently: Blockaid must either figure out a small set of queries to ask, a difficult problem, or fetch all accessible information, an expensive task. Second, Blockaid is designed for conventional applications that do not *rely on* an enforcer for data-access compliance. These applications should not be issuing queries that fail trace determinacy but pass instance-based determinacy: Such queries can, in Blockaid's absence, reveal inaccessible information on *another* database and typically indicate application bugs. Thus, Blockaid is right to flag them.

Theoretically optimal templates. While decision templates produced by Blockaid are general enough in practice, they might not be *maximally general* among all sound templates that match the query and trace being checked. For one thing, the template condition might not be maximally weak (§6.3.3). For another, a maximally general template can have a *longer* trace than the concrete one, a possibility Blockaid never explores.

Fundamentally, our template generation algorithm is limited by its *black-box access* to the policy: It interacts with the policy solely by checking template soundness using a solver. Producing maximally general templates might require opening up this black box and having the policy guide template generation more directly, a path we plan to explore in future work.

10 Conclusion

Blockaid enforces view-based data-access policies on web applications in a semantically transparent and backwards compatible manner. It verifies policy compliance using SMT solvers and achieves low overhead using a novel caching and generalization technique. We hope that Blockaid's approach will help rule out data-access bugs in real-world applications.

Acknowledgments

We are grateful to Alin Deutsch and Victor Vianu for the many discussions about query determinacy, and to Nikolaj Bjørner, Alvin Cheung, Vivian Fang, and members of the Berkeley NetSys Lab for their help with the project. We also thank the anonymous reviewers and our shepherd Malte Schwarzkopf for their helpful comments. This research was funded in part by NSF grants 1817116 and 2145471, and gifts from Intel and VMware.

References

[1] Foto N. Afrati. Determinacy and query rewriting for conjunctive queries and views. *Theor. Comput. Sci.*, 412(11):1005–1021, 2011.

[2] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *ICDE*, pages 1013–1022. IEEE Computer Society, 2005.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *VLDB*, pages 143–154. Morgan Kaufmann, 2002.

[4] Warwick Ashford. Facebook photo leak flaw raises security concerns, March 2015. https://www.computerweekly.com/news/2240242708/Facebook-photo-leak-flaw-raises-security-concerns.

[5] Autolab Project. https://autolabproject.com/.

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS*, 2022.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.

[8] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[9] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, pages 221–230. ACM, 2018.

[10] Gabriel Bender, Lucja Kot, and Johannes Gehrke. Explainable security for relational databases. In *SIGMOD*, pages 1411–1422. ACM, 2014.

[11] Gabriel Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. Fine-grained disclosure control for app ecosystems. In *SIGMOD*, pages 869–880. ACM, 2013.

[12] Alexander Brodsky, Csilla Farkas, and Sushil Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *IEEE Trans. Knowl. Data Eng.*, 12(6):900–919, 2000.

[13] Kristy Browder and Mary Ann Davidson. The virtual private database in Oracle9iR2. *Oracle Technical White Paper*, 2002.

[14] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14. ACM, 2013.

[15] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI*, pages 105–118. USENIX Association, 2010.

[16] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI*, pages 122–133. ACM, 2010.

[17] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *SOSP*, page 31–44. ACM, 2007.

[18] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proc. VLDB Endow.*, 11(11):1482–1495, 2018.

[19] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524. ACM, 2017.

[20] E. F. Codd. Relational completeness of data base sublanguages. In *Database Systems*. Prentice-Hall, 1972.

[21] Ellis S. Cohen. Information transmission in computational systems. In *SOSP*, pages 133–139. ACM, 1977.

[22] Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD*, pages 269–282. ACM, 2009.

[23] Leonardo de Moura. Z3 for Java. https://leodemoura.github.io/blog/2012/12/10/z3-for-java.html.

[24] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[25] Diaspora Foundation. The diaspora* project. https://diasporafoundation.org/.

[26] Django Software Foundation. Models | Django documentation | Django. https://docs.djangoproject.com/en/3.2/topics/db/models/.

[27] Tomasz Gogacz and Jerzy Marcinkowski. The hunt for a red spider: Conjunctive query determinacy is undecidable. In *30th Annual ACM/IEEE Symposium on Logic in*

Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015, pages 281–292. IEEE Computer Society, 2015.

[28] Tomasz Gogacz and Jerzy Marcinkowski. Red spider meets a rainworm: Conjunctive query finite determinacy is undecidable. In *PODS*, pages 121–134. ACM, 2016.

[29] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.

[30] Matthew Green. Twitter post: Piazza offers anonymous posting, but does not hide each user's total number of posts, October 2017. https://twitter.com/matthew_d_green/status/925053953330634753.

[31] Paolo Guagliardo and Leonid Libkin. A formal semantics of SQL queries, its validation, and applications. *Proc. VLDB Endow.*, 11(1):27–39, 2017.

[32] Marco Guarnieri and David A. Basin. Optimal security-aware query processing. *Proc. VLDB Endow.*, 7(12):1307–1318, 2014.

[33] Raju Halder and Agostino Cortesi. Fine grained access control for relational databases by abstract interpretation. In *ICSOFT*, volume 170, pages 235–249. Springer, 2010.

[34] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 319–347. IOS Press, 2012.

[35] Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[36] JRuby – the Ruby programming language on the JVM. https://www.jruby.org.

[37] Eddie Kohler. Hide review rounds from paper authors • kohler/hotcrp@5d53abc, March 2013. https://github.com/kohler/hotcrp/commit/5d53ab.

[38] Eddie Kohler. Download PC review assignments obeys paper administrators • kohler/hotcrp@80ff966, March 2015. https://github.com/kohler/hotcrp/commit/80ff96.

[39] Paraschos Koutris, Prasang Upadhyaya, Magdalena Balazinska, Bill Howe, and Dan Suciu. Query-based data pricing. In *PODS*, pages 167–178. ACM, 2012.

[40] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[41] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB*, pages 108–119. Morgan Kaufmann, 2004.

[42] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. STORM: refinement types for secure web applications. In *OSDI*, pages 441–459. USENIX Association, 2021.

[43] Mark H. Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple MUSes quickly. In *CPAIOR*, volume 7874 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2013.

[44] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2):223–250, 2016.

[45] Jorge Manrubia. jorgemanrubia/lazy_columns: Rails plugin that adds support for lazy-loading columns in active record models, 2015. https://github.com/jorgemanrubia/lazy_columns.

[46] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M. Frans Kaashoek, and Sam Madden. Towards multiverse databases. In *HotOS*, 2019.

[47] Mark Maunder. Vulnerability in WordPress Core: Bypass any password protected post. CVSS score: 7.5 (High), June 2016. https://www.wordfence.com/blog/2016/06/wordpress-core-vulnerability-bypass-password-protected-posts/.

[48] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. In *USENIX Security*, pages 1463–1479. USENIX Association, 2017.

[49] Microsoft. Row-level security - SQL Server, 2021. https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security.

[50] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241. ACM, 1999.

[51] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *ACM Trans. Database Syst.*, 35(3):21:1–21:41, 2010.

[52] Oracle. Using Oracle Virtual Private Database to control data access. https://docs.oracle.com/database/121/DBSEG/vpd.htm.

[53] Daniel Pasaila. Conjunctive queries determinacy and rewriting. In *ICDT*, pages 220–231. ACM, 2011.

[54] Alex Piechowski. Rails: How to use greater than/-less than in Active Record where statements, 2019. https://piechowski.io/post/how-to-use-greater-than-less-than-active-record/.

[55] Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In *AAAI*. AAAI Press, 2013.

[56] Giles Reger, Martin Suda, and Andrei Voronkov. Finding finite models in multi-sorted first-order logic. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 323–341. Springer, 2016.

[57] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark W. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2013.

[58] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, pages 551–562. ACM, 2004.

[59] Ruby on Rails Guides. Active Record basics. https://edgeguides.rubyonrails.org/active_record_basics.html.

[60] Luc Segoufin and Victor Vianu. Views and queries: determinacy and rewriting. In *PODS*, pages 49–60. ACM, 2005.

[61] Jie Shi, Hong Zhu, Ge Fu, and Tao Jiang. On the soundness property for SQL queries of fine-grained access control in DBMSs. In *ICIS*, pages 469–474. IEEE Computer Society, 2009.

[62] Software Freedom Conservancy. SeleniumHQ: Browser automation, 2021. https://www.selenium.dev/.

[63] Spree Commerce - a headless open-source ecommerce platform. https://spreecommerce.org/.

[64] Ben Stock. Search leaks hidden tags • Issue #135 • kohler/hotcrp, June 2018. https://github.com/kohler/hotcrp/issues/135.

[65] Michael Stonebraker and Eugene Wong. Access control in a relational data base management system by query modification. In *ACM Annual Conference*, pages 180–186. ACM, 1974.

[66] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic query exploration. In *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 49–68. Springer, 2009.

[67] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. Qex: Symbolic SQL query explorer. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 425–446. Springer, 2010.

[68] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*, pages 452–466. ACM, 2017.

[69] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. On the correctness criteria of fine-grained access control in relational databases. In *VLDB*, pages 555–566. ACM, 2007.

[70] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.*, 2(POPL):56:1–56:29, 2018.

[71] Zhiheng Wang. Navigation timing. W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-navigation-timing-20121217.

[72] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *PLDI*, pages 631–647. ACM, 2016.

[73] Wen Zhang, Eric Sheng, Michael Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. Blockaid: Data access policy enforcement for web applications, 2022. https://arxiv.org/abs/2205.06911.

[74] Zheng Zhang and Alberto O. Mendelzon. Authorization views and conditional query containment. In *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 259–273. Springer, 2005.

A Artifact Appendix

Abstract

Our artifact includes our Blockaid implementation, which is compatible with applications that can run atop the JVM and connect to a database via JDBC (§7). We also provide the three applications we used for our evaluation—modified according to §8.2—as well as the data-access policy we wrote for each. Finally, we provide a setup for reproducing the evaluation results from §8.

Table 3: *Where artifact contents are hosted.*

Content	Location	Branch / tag / release
Artifact README	https://github.com/blockaid-project/artifact-eval	`main` branch
Blockaid source	https://github.com/blockaid-project/blockaid	`main` branch (latest version) `osdi22ae` branch (AE version)[a]
Experiment launcher	https://hub.docker.com/repository/docker/blockaid/ae	`latest` tag
Launcher source	https://github.com/blockaid-project/ae-launcher	`main` branch
VM image	https://github.com/blockaid-project/ae-vm-image	`osdi22ae` release
Experiment scripts	https://github.com/blockaid-project/experiments	`osdi22ae` branch
Applications		
diaspora*	https://github.com/blockaid-project/diaspora	`blockaid` branch[b]
Spree	https://github.com/blockaid-project/spree	`bv4.3.0-orig` branch (original)[c] `bv4.3.0` branch (modified)[d]
Autolab	https://github.com/blockaid-project/Autolab	`bv2.7.0-orig` branch (original)[c] `bv2.7.0` branch (modified)[d]
Policies for applications	https://github.com/blockaid-project/app-policies	`main` branch

[a] The "AE version" is the version of Blockaid used in artifact evaluation.
[b] The same diaspora* branch is used for both baseline and Blockaid measurements. The code added for Blockaid is gated behind conditionals that check whether Blockaid is in use.
[c] "(original)" denotes the original application modified only to run on top of JRuby.
[d] "(modified)" denotes the "(original)" code additionally modified to work with Blockaid (§8.2).

Scope

This artifact can be used to run the main experiments from this paper: the page load time (PLT) measurements (§8.4) and the fetch latency measurements (§8.5 and §8.6) on the three applications. From these experiments, it generates Table 2 (with URLs and descriptions omitted), Figure 2, and Figure 3. Because the full experiment can be time- and resource-consuming (taking roughly 15 hours on six Amazon EC2 c4.8xlarge instances), the experiment launcher can be configured to take fewer measurement rounds at the expense of accuracy.

Our Blockaid implementation can also be used to enforce data-access policies on new applications, as long as they have been modified to satisfy our requirements (§3.3), run atop the JVM, and connect to the database using JDBC (§7).

Contents

This artifact consists of our Blockaid implementation, the three applications used in our evaluation (with modifications described in §8.2), the data-access policy we wrote for each, and scripts and virtual machine image for running the experiments.

Hosting

See Table 3.

Requirements

The experiment launcher, which relies on Docker, launches experiments on Amazon EC2 and so requires an AWS account. By default, it uses six c4.8xlarge instances—to run the PLT and fetch latency experiments for the three applications simultaneously. However, it can be configured to launch fewer instances at a time (e.g., to run the experiments serially, using one instance at a time).

SHORTSTACK: Distributed, Fault-tolerant, Oblivious Data Access

Midhul Vuppalapati* Cornell University
Kushal Babel* Cornell University
Anurag Khandelwal Yale University
Rachit Agarwal Cornell University

Abstract

Many applications that benefit from data offload to cloud services operate on private data. A now-long line of work has shown that, even when data is offloaded in an encrypted form, an adversary can learn sensitive information by analyzing data access patterns. Existing techniques for oblivious data access—that protect against access pattern attacks—require a centralized and stateful trusted proxy to orchestrate data accesses from applications to cloud services. We show that, in failure-prone deployments, such a centralized and stateful proxy results in violation of oblivious data access security guarantees and/or in system unavailability. We thus initiate the study of distributed, fault-tolerant, oblivious data access.

We present SHORTSTACK, a distributed proxy architecture for oblivious data access in failure-prone deployments. SHORTSTACK achieves the classical obliviousness guarantee—access patterns observed by the adversary being independent of the input—even under a powerful passive persistent adversary that can force failure of arbitrary (bounded-sized) subset of proxy servers at arbitrary times. We also introduce a security model that enables studying oblivious data access with distributed, failure-prone, servers. We provide a formal proof that SHORTSTACK enables oblivious data access under this model, and show empirically that SHORTSTACK performance scales near-linearly with number of distributed proxy servers.

1 Introduction

Cloud services offer applications scalable, fault-tolerant, and easy-to-manage systems for storing and querying data. Many applications that benefit from offloading data to these cloud services operate on private data that can reveal sensitive information even when stored in an encrypted form [1–6]. An example is that of medical practices offloading patient health data to the cloud [7–9]—charts accessed by oncologists can reveal not only whether a patient has cancer, but also depending on the frequency of accesses (*e.g.*, the frequency of chemotherapy appointments), indicate the cancer's type and severity. Several such applications are subject to severe security concerns.

*Equal contributions.

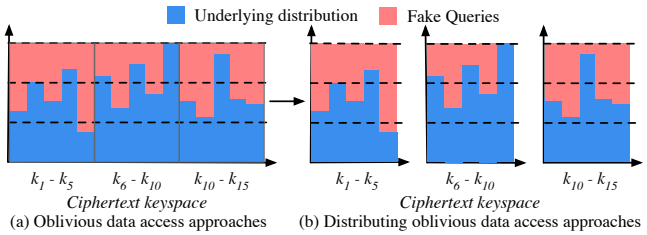
There is a large and active body of research on building systems for oblivious data access, that is, hiding not only the content of the data, but also data access patterns (*e.g.*, access frequency across data objects). These systems use one of the two techniques—Oblivious RAM [10–17] that enables oblivious data access against active adversaries but has bandwidth overheads that are logarithmic in the number of data objects, or Pancake [6, 18, 19] that enables oblivious data access against passive persistent adversaries with a small constant bandwidth overhead. Both of these techniques provide a powerful oblivious data access guarantee: an adversary observing all queries to and all responses from the cloud storage service observes uniform random accesses over the encrypted data objects. The challenge, however, is that both of these techniques require a centralized, *stateful*, proxy to orchestrate data access from applications to cloud services. Such a centralized and stateful proxy means that existing systems for oblivious data access suffer from two core issues (§3.1):

- *Security violation, or long periods of system unavailability during proxy failures:* The proxy being stateful means that, upon a failure, the proxy may lose state. We show in §3.1 that, if the proxy state is lost, naïvely restarting a new proxy and executing queries without restoring the state would lead to violation of oblivious data access security guarantees. To avoid such a security violation, upon restarting a new proxy, the state must be restored before executing any queries, *e.g.*, by downloading the entire data and metadata from the cloud, decrypting all the data, reconstructing the (ORAM or Pancake) data structure, re-encrypting all the data, and uploading all the data back to the storage service; this would lead to long periods of system unavailability.
- *Bandwidth and/or compute scalability bottlenecks:* Since the proxy receives multiple responses for each client query, it has bandwidth overheads ($\Omega(\log n)$ in ORAM [20–25] and $3\times$ in Pancake [6]); and, since the proxy is responsible for both data encryption/decryption and processing for each individual query and response, it has non-trivial compute overheads. Thus, the centralized proxy can become bandwidth or compute bottlenecked, limiting system throughput.

We present SHORTSTACK, a distributed, fault-tolerant, system for oblivious data access. SHORTSTACK achieves three desirable goals: (1) formal oblivious data access guarantee against passive persistent adversaries, even under failures; (2) system availability even when an arbitrary, bounded-sized, subset of distributed proxy servers may fail; and (3) near-linear throughput scalability with number of distributed proxy servers. In designing SHORTSTACK, we make three core contributions.

Our first contribution is to fundamentally establish security goals for oblivious data access in failure-prone deployments. Indeed, existing security models [6, 10–17, 26] do not capture failures. We introduce a formal security model and a security definition to study distributed, fault-tolerant, systems for oblivious data access under passive persistent adversaries. The model requires the classical oblivious data access guarantee [6, 10]—access patterns observed by the adversary must be independent of the input; in addition, to capture failures, the model requires this guarantee to hold under a powerful adversary that can fail an arbitrary (bounded-sized) subset of distributed proxy servers at arbitrary times. Informally, under our security definition, a scheme is considered secure if the access distribution over encrypted data objects is independent of the input distribution, even with adversarial choice and time of proxy server failures.

Our second contribution is design of a distributed, fault-tolerant, proxy architecture—SHORTSTACK—that enables oblivious data access against passive persistent adversaries, system availability (under a bounded number of failures), and near-linear throughput scalability with number of proxy servers. Simultaneously guaranteeing these three properties, especially when proxy servers can fail, turns out to be challenging: to avoid bandwidth and compute bottlenecks, multiple proxy servers must simultaneously process and send queries to the storage server; this makes it non-trivial, if not impossible, to ensure uniform random access over encrypted objects at all times (*e.g.*, right after one of the proxy server fails) without giving up on availability. The key insight in SHORTSTACK design is that obliviousness only necessitates that access patterns observed by the adversary are independent of the input; the requirement of uniform random access over all encrypted objects as in prior designs is one, but not the only, way to achieve such independence. SHORTSTACK design achieves such independence as follows. The security of oblivious data access techniques stems from "flattening" the access distribution over unencrypted (plaintext) objects to a uniform random one over encrypted (ciphertext) objects (Figure 1 (a)). As illustrated visually in Figure 1 (b), any uniform random distribution over ciphertext objects can be decomposed into multiple sub-distributions in a manner that (1) each sub-distribution is uniform random over its support; and (2) the set of objects in any sub-distribution is equal in size, disjoint, and random. Thus, if each proxy server that forwards queries to the storage server is responsible for one of the sub-distributions, even with failure of a subset of these



(a) Oblivious data access approaches (b) Distributing oblivious data access approaches

Figure 1: The flattened distribution over all ciphertext keys in oblivious data access schemes can be expressed as a sum of distributions over disjoint subsets of ciphertext keys.

proxy servers, the adversary observes nothing but a uniform distribution (using (1)) over a random subset (using (2)) of objects. Achieving independence, and not necessarily a uniform random access pattern, at all times is at the core of the SHORTSTACK design. In §4, we present a novel layered SHORTSTACK architecture that, using k physical proxy servers, maintains system availability with up to $(k-1)$ proxy server failures and achieves throughput a factor $\sim k$ higher than a single proxy, all while enabling oblivious data access.

Our third contribution is a formal proof that SHORTSTACK enables oblivious data access under the above security model, and empirical evidence that SHORTSTACK can achieve near-perfect scalability with number of proxy servers (assuming storage server is not the bottleneck). We also show that SHORTSTACK gracefully handles failures: in the worst-case, SHORTSTACK throughput reduces linearly with number of proxy server failures (as one would expect). For the current SHORTSTACK implementation, the cost of achieving oblivious data access, availability and scalability is a $\sim$7ms increase in latency, a tiny fraction of the usual wide-area network latency. An end-to-end implementation of SHORTSTACK is available at https://github.com/pancake-security/shortstack.

2 SHORTSTACK Background

We describe our system, failure, and threat models, followed by a brief primer on oblivious data access approaches.

2.1 System, Threat and Failure Models

System model. We consider settings where applications offload data to the cloud to benefit from the many properties enabled by cloud services, *e.g.*, strong data durability and persistence, geo-replication, lower cost than provisioning dedicated and replicated storage servers, transparent handling of devices wearing out, and others. Examples of such applications include cloud-based healthcare services [9, 27–29] as well as classical applications from access pattern attack literature [6, 11]. The cloud-based storage service implements a key-value (KV) store that stores a collection of KV pairs, and support the following single-key operations: get, put, and delete. SHORTSTACK design can be applied to any data store that supports single-key read/write/delete operations.

SHORTSTACK employs the standard encryption proxy model, commonly used in encrypted data stores [6, 15, 16,

30–35]: a trusted proxy orchestrates query execution from one or more client applications; the only difference compared to previous designs is that, in SHORTSTACK architecture, the proxy is logically-centralized but physically-distributed—that is, client queries may now be routed though multiple physical proxy servers within the same trusted domain.

All network channels are encrypted using TLS. Each key k in the KV store is encrypted using a pseudorandom function (PRF), denoted by $F(k)$; each value v is symmetrically encrypted, denoted by $E(v)$. The logically-centralized proxy stores secret cryptographic keys needed for F and E, and performs encryption. Since F is deterministic, the proxy can execute all queries related to key k by sending $F(k)$ to the cloud service. Similar to many existing commercial deployments [31–35], keys and values are padded to a fixed size to avoid any length-based leakage.

Threat model. SHORTSTACK builds upon the widely-used trusted proxy threat model [11–13, 15], where one or more mutually-trusting clients execute operations on an untrusted cloud storage service via a trusted proxy; as mentioned earlier, the only difference in SHORTSTACK is that the proxy is logically-centralized but comprises physically-distributed servers. As in many prior works [6, 11, 30], we consider scenarios where the clients and the proxy servers all belong to a trusted domain. The storage service is controlled by an honest-but-curious (or, a passive persistent) adversary that observes all encrypted accesses but does not actively perform its own accesses. Since network channels are encrypted using TLS, the adversary cannot observe communications within the trusted domain, that is, the adversary cannot observe traffic between the clients and proxy servers.

We model queries to the KV store using the Pancake model [6]: queries are generated as a sequence of accesses sampled from a (time-varying) distribution π over n KV pairs. While the encryption mechanism has an estimate of the distribution $\hat{\pi}$, the adversary knows both the distribution π and the transcript of encrypted queries and responses. We define a formal security model and definition in §5, but informally, the system is secure if the transcript is *independent* of the underlying distribution π, i.e., the adversary cannot identify an association between the two.

Failure model. We assume the cloud service provides data durability. However, proxy servers can fail. We consider the fail-stop model [36] for proxy server failures.

2.2 Oblivious Data Access Approaches

There are two approaches to oblivious data access today—the classical ORAM [10–17], and the more recent approach of frequency smoothing as in Pancake [6, 18, 19]. ORAMs are designed to prevent a broad range of attacks (*e.g.*, active adversaries); accordingly, they also suffer from high overheads, *e.g.*, recent results [20–25] have established strong lower bounds on ORAM overheads—for a data store with n KV pairs, any

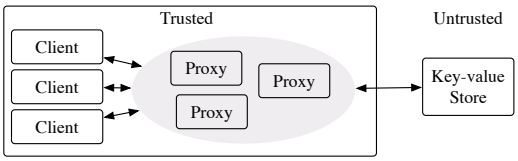


Figure 2: SHORTSTACK System and Threat model

ORAM design must incur bandwidth overheads of $\Omega(\log n)$ (for proxy storage sublinear in KV store size). For KV stores that store millions or billions of KV pairs, these overheads may amount to orders-of-magnitude of throughput loss [6, 37], making ORAMs impractical. Pancake enables oblivious data access against passive persistent adversaries, and incurs a small, constant, bandwidth overhead of $3\times$, independent of the number of objects in the KV store. Thus, we focus on building a distributed, fault-tolerant, proxy architecture within the Pancake context. To keep the paper self-contained, we summarize the Pancake mechanisms necessary to understand the SHORTSTACK architecture.

A brief primer to oblivious data access using Pancake. The Pancake approach combines the knowledge of the distribution estimate $\hat{\pi}$ with several techniques (selective replication, fake accesses, batching, etc.) to transform a sequence of queries into uniform accesses over encrypted KV pairs. Selective replication creates "replicas" of KV pairs that have high access probability relative to other KV pairs, which serves to partially smooth the distribution over (replicated) KV pairs, while also ensuring that the total number of keys to be stored in the KV store is exactly $2n$ (if needed, dummy replicas are added so that the number of replicas does not reveal any distribution-sensitive information). To hide the association between the original keys and their replicas, each replica (k,i) of an unencrypted key k is protected by applying the pseudorandom function F, discussed in §2.1, to the replica identifier to generate an encrypted label $F(k,i)$ for the replica. In the rest of the paper, we refer to the original unencrypted key as the plaintext key, and the encrypted label for each replica as the ciphertext key. To remove the remaining non-uniformity, "fake" queries are added: these queries are sampled from a carefully crafted fake access distribution π_f to boost the likelihood of accessing replicated KV pairs, until the resulting distribution is uniform.

Security requires ensuring that fake and real queries be indistinguishable; to achieve this, encrypted queries are issued in small batches of size B, where each query is either real or fake with equal probability. Since the adversary cannot observe traffic between the clients and the proxy server, it has no way to distinguish real and fake queries within any batch. To prevent an adversary from distinguishing between reads and writes, every access is performed as a read followed by write of a freshly encrypted value. Writes to keys with multiple replicas could reveal which replicas belong to the same key; thus, only one replica is updated at the time of the write query, and the write value is cached at the proxy in

a data structure called the UpdateCache, and the remaining replicas are opportunistically updated during subsequent fake or real queries to the replicas.

Dynamic adaptation to changes in the underlying access distribution is achieved by adjusting the fake-distribution (π_f), and by reassigning the number of replicas across keys. This can be done securely by exploiting the observation that the total number of replicas is exactly $2n$, regardless of the underlying distribution. As such, when the distribution changes, for every key that must lose a replica, another must gain a replica to ensure the distribution remains smooth. Thus, replicas can be reassigned opportunistically for all such key-pairs using a replica-swapping protocol.

In summary, to enable oblivious data access for the general case of read/write workloads and for time-varying distributions, Pancake uses a centralized, *stateful*, proxy that stores (1) the UpdateCache to buffer writes until they are opportunistically propagated to all the replicas; (2) distribution-related state; and (3) replica-related state, to execute the replica swapping process during distribution changes. Using this state, Pancake enables oblivious data access by performing three tasks at the proxy in failure-free scenarios: (1) generating "fake" queries for each real client query; (2) updating UpdateCache upon each query; and (3) issuing a batch of queries comprising real and fake queries to the server, and relaying the response for the real query back to the client.

3 Limitations of Strawman approaches

In this section, we describe subtle security vulnerabilities with strawman approaches to designing distributed, fault-tolerant, systems for oblivious data access.

3.1 Centralized proxy: Insecure and/or long periods of unavailability

The stateful nature of the centralized proxy makes it challenging to simultaneously achieve oblivious data access security guarantees, availability and scalability upon a failure. If achieving scalability were the only goal, the proxy server could be overprovisioned with large bandwidth and/or compute resources; however, achieving security and availability upon a failure is hard due to the proxy being stateful: the naïve solution of replacing the failed proxy server with a new one and having clients reissue failed queries results in violating security and correctness guarantees:

- Consider the (simplest) case of a read-only workload with a static access distribution. Replacing a failed proxy server with a new one, and having clients reissue the failed queries, results in the following subtle security issue. Consider a real query on key k; and consider the scenario where the proxy fails in the middle of sending out queries (both real and fake) in the batch to the KV store, that is, some of the queries in the batch have been sent out while others are lost. Since the proxy has failed, the client would receive no response for k; thus, upon restarting the proxy, the client will retry a real query on k. The retried queries will result in the same real accesses, but potentially new fake accesses. An adversary can thus exploit the transcript of queries at the server to identify real queries with high confidence by isolating repeated accesses right before and right after a failure, hence gaining sensitive information.
- Write queries make the problem significantly more challenging. Consider a write query to a key with two replicas; suppose the proxy fails when the write value has propagated to only one of the replicas (and thus, is buffered in the UpdateCache waiting to be propagated to the other replica). We now replace the failed proxy with a new one. Since the UpdateCache state is lost, when a read query for this key is received, the new proxy could end up reading the value from one of the stale replicas, violating the data correctness/consistency guarantee. Alternatively, if the new proxy reads all replicas of the key to determine which one has the latest value (*e.g.*, using timestamps), oblivious data access guarantees would be violated since an adversary can identify replica correlations (replicas being accessed belong to the same key) by analyzing queries right after a failure.

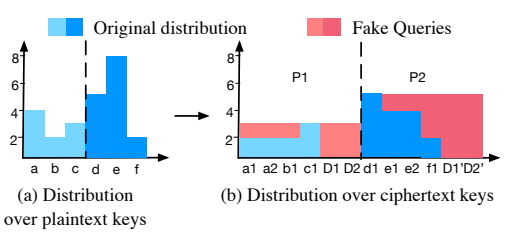
For a centralized stateful proxy design and for the general case of read/write workloads over time-varying distributions, to avoid the above security and correctness violations upon a failure, the proxy state must be reconstructed—*e.g.*, by downloading all the data from the cloud service, decrypting the data, reinitializing the data structures, re-encrypting the new data structures, and writing all the new data back to the server—before issuing new queries. Even for moderate-sized KV stores, this would incur extremely large bandwidth and compute overheads, as well as long unavailability periods.

In summary, replacing the centralized proxy server with a new one (upon a failure) and having clients reissue the queries either fails to ensure critical system properties (security and/or correctness), or results in large unavailability periods. This motivates the need for a distributed proxy architecture.

3.2 Challenges in Distributing Proxy Logic

We now describe security and correctness vulnerabilities with naïvely distributing the proxy state and logic across multiple physical servers.

Naïvely partitioning both the proxy state and the query execution responsibility leads to security violations. A straightforward approach to designing a distributed proxy for oblivious data access is to partition both the proxy state and the query execution responsibility across multiple physical servers—each proxy server stores the UpdateCache and access distribution for a subset of the plaintext keys (*e.g.*, using hash partitioning over the plaintext keys); clients forward their (real) query on key k to the proxy server responsible for k; and, upon receiving a real query, the proxy server generates fake queries based on distribution *corresponding to its own partition*, and executes these queries on the storage service.



(a) Distribution over plaintext keys

(b) Distribution over ciphertext keys

Figure 3: Security violation in one-layer approach.

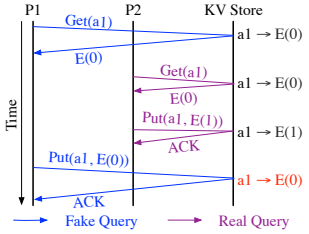
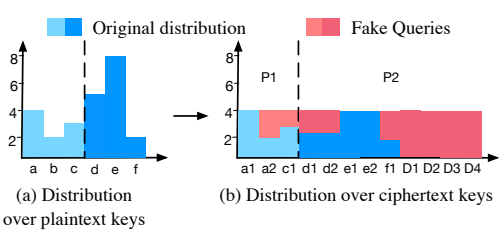


Figure 4: Correctness violation in one-layer approach.



(a) Distribution over plaintext keys

(b) Distribution over ciphertext keys

Figure 5: Security violation in two-layer approach.

While this approach scales linearly with number of physical proxy servers, it suffers from security vulnerability. In particular, it does not guarantee that the resulting distribution observed by the adversary is independent of the input distribution. Consider the scenario shown in Figure 3 (a). Here, plaintext keys are partitioned across two proxy servers—P1 is responsible for keys $\{a, b, c\}$, and P2 is responsible for keys $\{d, e, f\}$. Since, each proxy server operates only on its local plaintext key partition, P1 selectively replicates key a into 2 replicas a_1, a_2, and introduces two dummy key replicas D_1, D_2, leading to a total of 6 ciphertext keys; it then adds fake queries to make the access distribution across these ciphertext keys uniform. Similarly, P2 selectively replicates key e into 2 replicas e_1, e_2, and introduces two dummy key replicas D'_1, D'_2, again leading to a total of 6 ciphertext keys; P2 then adds fake queries to make the access distribution across these ciphertext keys uniform. Figure 3 (b) shows the final output access distribution over ciphertext keys. Since P1 and P2 smooth the distribution over their sets of plaintext keys independently, and since the key set assigned to P2 has a higher average access frequency than the key set assigned to P1, the frequency of accesses over ciphertext keys for P2 is higher than the frequency of accesses over ciphertext keys for P1. In particular, the overall access distribution over all ciphertext keys is dependent on the input distribution over the two subset of keys, thus leaking sensitive information.

Replicating proxy state across all physical servers but naïvely partitioning query execution responsibility leads to security violations. To avoid the security vulnerability in the previous scenario, one possible approach is to replicate the entire proxy state (UpdateCache and access distribution) across all physical servers in the distributed proxy. We will need to keep the state consistent across all physical servers—various mechanisms exist for this; for instance, clients can broadcast each query to each physical server to keep the access distribution consistent, and servers could use a distributed protocol (*e.g.*, state machine replication) to keep the UpdateCache consistent. Let us ignore the scalability issues with maintaining such consistent state for a moment.

To avoid bandwidth and compute bottleneck, we still want each query to be executed at one (or a small number) of the physical proxy servers. Thus, each physical server will now be responsible for receiving real queries from the clients for a subset of the keys (again, *e.g.*, using hash partitioning over the plaintext keys), and generating fake queries for each real query (now on the entire distribution). One question remains: which physical server should send the (real and fake) queries to the storage service on the cloud? Unfortunately, both the obvious solutions—the server generating the batch executes all queries in the batch, and the server responsible for plaintext key k executes all (real and fake) queries for the key k (independent of which server generated the fake query)—suffer from security and/or correctness vulnerabilities.

To see the issue with the first solution, consider the example in Figure 4 with two proxy servers P1 and P2: to serve a client query to write value 1 to key a, P2 sends a get($F(a,1)$) followed by put($F(a,1)$, $E(1)$) query to the KV store, where $(a,1)$ is one of the ciphertext key, or replica, corresponding to a. At the same time, P1, unaware of P2 ongoing write query, sends a fake put query to the same ciphertext key $(a,1)$ in response to another client query. Based on the timeline of operations shown in Figure 4, the fake put from P1 overwrites the real put from P2, resulting in incorrect system behavior. Note that the incorrectness occurs since two different proxy servers issue queries for the same ciphertext key.

Unfortunately, the second solution also suffers from security vulnerabilities—partitioning the query execution across physical servers reveals not only which *plaintext* keys are managed by each server, but also their relative access frequencies. Figure 5 shows an example; the scenario is the same as Figure 3, but with selective replication and fake query generation done over the entire distribution across all plaintext keys—thus, as shown in Figure 5 (b), in addition to selective replication of keys d and e, 4 dummy key (D) replicas (D_1, D_2, D_3, D_4) were added, and the access distribution across ciphertext keys is uniform. We use the same partitioning of plaintext keys across P1 and P2 as in the example of Figure 3—P1 handles all real and fake queries for the three less popular plaintext keys, while P2 handles all queries for the three more popular plaintext keys and the dummy key. The challenge, however, is that although each server handles roughly equal number of plaintext keys, the number of ciphertext keys handled by P1 ($= 3$) and P2 ($= 9$) are very different. This leaks the subset of keys handled by each server and, by extension, their relative popularities to the adversary. Even if the volume of traffic issued by individual proxy servers is hidden (*e.g.*, via a trusted gateway/NAT so that all proxy servers have the same publicly visible IP address), failures of one of the physi-

cal proxy servers would reveal the same information. Even if clients were to retry their queries upon a failure, in-flight queries to the KV store from a failed server would be repeated, again revealing the same information.

Summary. The above discussion leads to three different design principles for distributed, fault-tolerant, oblivious data access systems. From the partitioning-based approach, we learn that, to achieve oblivious data access, each physical server in the distributed proxy should perform selective replication and (fake) query generation over the entire distribution across all plaintext keys (thus, each physical server should know the access distribution across the entire set of plaintext keys). The replication-based approach leads to two additional principles. First, even if proxy state can be replicated in a consistent and scalable manner, maintaining correctness requires that no two physical proxy servers should send the queries for the same ciphertext key; in other words, query execution should be partitioned by ciphertext keys across different physical servers. Second, to avoid security vulnerability, no single proxy server should be deterministically responsible for executing queries for all ciphertext keys corresponding to the same plaintext key; that is, query execution should be partitioned by ciphertext keys—randomly, and independent of plaintext keys—across physical proxy servers.

4 SHORTSTACK Design

We now present the SHORTSTACK distributed, fault-tolerant, proxy architecture.

4.1 Design Overview

SHORTSTACK uses a novel layered architecture, with three *logical* layers*, as shown in Figure 6. Each layer has multiple logical proxy servers for fault tolerance and/or scalability purposes, and embodies one of the three design principles outlined at the end of the previous subsection. In the first layer (L1), proxy servers are responsible for a random subset of client queries—upon receiving a real client query on a plaintext key, the server generates real and fake queries (over ciphertext keys); importantly, fake queries are generated using the *entire* access distribution across all plaintext keys. In the second layer, L2, proxy servers are responsible for maintaining a partition of the UpdateCache state; importantly, the UpdateCache is partitioned by *plaintext* keys across the L2 servers. Finally, in the third layer, L3, each proxy server is responsible to execute real and fake queries on the KV store for a random, distinct, subset of *ciphertext* keys.

We outline the lifetime of a query with the layered SHORTSTACK architecture in a *failure-free scenario*. The client sends the query to a randomly selected L1 proxy server; the L1 server generates the batch comprising real and fake queries (recall, these generated queries are on ciphertext keys). The L1 server then forwards each individual query within the batch to

*On a lighter note, our work seems to formally establish the widely-agreed belief that three is the right number for a SHORTSTACK [38]!

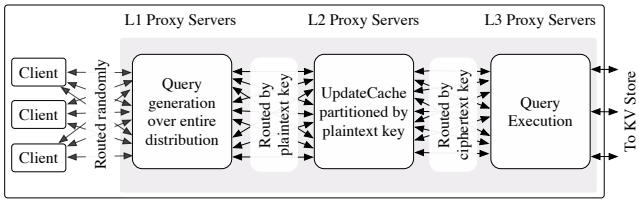


Figure 6: An overview of three-layer SHORTSTACK architecture

one of the L2 servers—the one that maintains UpdateCache state for the corresponding plaintext key in the query. Upon receiving a query, an L2 server updates its local partition of the UpdateCache, and forwards the query to one of the L3 servers—the one that is responsible for executing queries for that ciphertext key. The L3 server ultimately forwards the query to the KV store; upon receiving a response from the KV store, the L3 server sends a response for the real query back to the client, as well as an acknowledgement in the reverse direction of the original path taken by the query (from L3 to L2 to L1) to clear any buffered state associated with the query. We provide more details on the three-layer SHORTSTACK architecture and query execution in §4.2.

For fault-tolerance against f failures, each of the L1 and L2 proxy servers use $f+1$ replication along with the chain replication protocol [39]. Replicating L1 servers prevents the security vulnerabilities discussed in §3.1 that are caused by clients retrying queries upon failures. Specifically, as we discuss in §4.3, SHORTSTACK uses chain replication to guarantee that a batch of queries is never partially executed—either all the queries in a batch are (eventually) forwarded to the KV store or none of them are—thus preventing access pattern leakage even when failures happen. Replicating L2 servers prevents UpdateCache state from being lost due to failures. As we will show, L3 server failures do not have the same security vulnerability as L1 and L2 server failures. Thus, L3 layer is not chain replicated; however, it needs at least $f+1$ servers to maintain availability during failures—if one of the L3 server fails, the remaining L3 servers take over its load. Upon an L3 server failure, in-flight queries at the server will be dropped and L2 servers will reissue the dropped queries. While this results in duplicate queries being forwarded to the KV store, we will show that these duplicate queries do not reveal any sensitive information—the adversary would only observe duplication of queries to a random subset of labels independent of the input distribution. We provide more details on SHORTSTACK mechanisms for handling failures in §4.3.

SHORTSTACK design allows independently setting desired fault tolerance f and scalability factor k. Specifically, to achieve a factor k scalability—that is, achieving system throughput a factor of k higher than a centralized proxy—along with fault tolerance against f failures, SHORTSTACK creates k independent L1 and L2 chains that operate in parallel. The case of L3 is again different; if $f+1 > k$, SHORTSTACK will already have at least k L3 proxy servers to ensure fault tolerance (as described earlier). For $f+1 \leq k$, SHORTSTACK

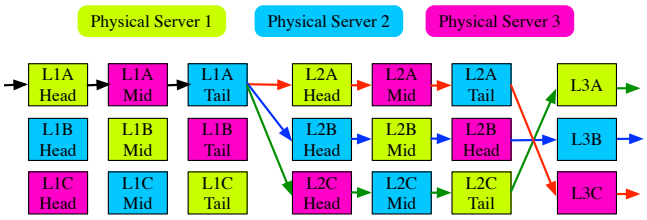


Figure 7: An instantiation of SHORTSTACK's three-layer architecture that guarantees system security and availability with $f = 2$ failures, and achieves $k = 3\times$ scalability (as defined in §4.1). Multiple logical layers in SHORTSTACK are colocated on the same physical server. The arrows depict the lifetime of a single query.

uses a total of k L3 proxy servers, thus guaranteeing both fault tolerance against f failures and a factor k scalability. Figure 7 shows an example for $f = 2$ and $k = 3$.

Colocating SHORTSTACK logical layers and their replicas on a small number of physical servers. Consider an instantiation of SHORTSTACK with fault-tolerance against up to $f = 2$ failures and $k = 3$ scalability. Then, given the above design, SHORTSTACK will require 3 L1 and 3 L2 chains, each having 3 logical replicas within the chain replication protocol. In addition, SHORTSTACK will require 3 logical servers in the L3 layer. Overall, for $f = 2$ and $k = 3$, SHORTSTACK requires 21 "logical" units. However, as shown in Figure 7, all these 21 logical units can be packed on 3 physical servers without compromising security, fault tolerance, availability and scalability. In particular, the replicas of each logical server in each layer are staggered across the physical servers such that no two replicas of the same logical server within the same layer are co-located on the same physical server. Hence, even upon failure of any two physical servers, one replica from each of the L1 servers, one replica from each of the L2 servers, and one L3 server will still be alive, ensuring security, availability and $f = 2$ fault tolerance. In general, using a technique from [40], SHORTSTACK achieves a factor k scalability and fault tolerance against $f \leq k-1$ failures, using only k physical servers. Since any system that tolerates f failures and achieves a factor k scalability must require at least $\max(f+1,k)$ physical servers, SHORTSTACK uses minimum resources to provide these properties.

We provide more details on design of each layer in the SHORTSTACK layered architecture, the mechanisms for fault tolerance, and the mechanisms for handling dynamic distributions in §4.2, §4.3 and §4.4, respectively.

4.2 SHORTSTACK Design Details

In this subsection, we describe SHORTSTACK's three-layer architecture in detail, for the case of no failures and static access distribution. We will extend this design to handle failures and dynamic distributions in §4.3 and §4.4, respectively.

In a failure-free scenario, the key challenge that SHORTSTACK addresses relative to a single proxy architecture is scalability. To achieve k-factor scalability in the failure-free scenario, SHORTSTACK uses k (logical) proxy servers in each layer. For example, in Figure 7, each layer would consist of three nodes, *e.g.*, L1A, L1B and L1C for the L1 layer, L2A, L2B and L2C for the L2 layer, and L3A, L3B and L3C for the L3 layer.

Details of three-layer operation. Figure 8 details the precise initialization and L1/L2/L3 server logic in SHORTSTACK. SHORTSTACK employs the following functionalities from PANCAKE ($\mathcal{P}$) [6] as a black-box:

- an Init function, which takes as input an estimate of the access distribution $\hat{\pi}$ and the unencrypted KV store KV of size n plaintext keys, and generates an encrypted KV store KV$'$ of size $2n$ ciphertext keys, along with a fake distribution π_f over KV$'$;
- a Batch function, which takes a query on a plaintext key k in KV as input, and generates (using $\hat{\pi}$ and π_f) a batch of B ($B = 3$ by default) ciphertext queries to KV$'$; and,
- an UpdateCache function that internally updates per-plaintext key state, and returns an encrypted (possibly updated) value to be written to the KV store.

We now outline how SHORTSTACK distributes the execution of PANCAKE across its three-layer design:

*Initialization (*Init() *in Figure 8):* SHORTSTACK first performs PANCAKE initialization (using $\mathcal{P}$.Init), transforming the unencrypted KV store KV with n plaintext keys to the encrypted KV store KV$'$ using $2n$ ciphertext keys, using an estimate of the underlying access distribution $\hat{\pi}$. During the process, the adversary just observes insert operations of $2n$ ciphertext keys, which does not reveal any information. SHORTSTACK then initializes and configures k logical proxy servers in each of the three layers on top of k physical servers. Finally, SHORTSTACK computes a weight vector δ, containing weights assigned to each L2 server (proportional to the volume of ciphertext traffic generated by it). As will be discussed, L3 servers use these weights to process L2 queries such that the queries issued by L3 servers appear uniform random (recall, this subsection focuses on failure-free scenario, where SHORTSTACK achieves uniform random distribution over ciphertext keys).

Query processing logic (s_{L1}.ProcessQuery(), s_{L2}.Process() *and* s_{L3}.Process() *in Figure 8):* Clients forward each query to a randomly chosen L1 proxy server. Upon receiving a query, the L1 server generates a batch of B queries (using $\mathcal{P}$.Batch) that comprises both real and fake queries to KV$'$. The L1 server then enqueues each query in the batch across different L2 servers based on the hash of the query's plaintext key.

Upon receiving a query, an L2 server calls $\mathcal{P}$.UpdateCache which leads to two sequential actions. First, the per-plaintext key state stored at the L2 server is updated; and second, if this query can be used to propagate outstanding write requests into the plaintext key replicas, the value to be written to the KV store is updated. It then forwards the query to the corresponding L3 server based on the hash of the query's ciphertext key (denoted as "Enqueue" in Figure 8).

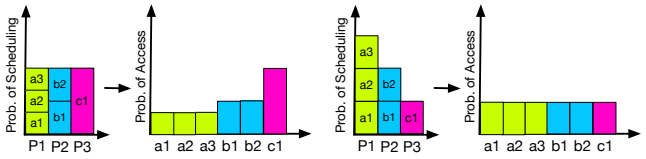
```
Init(π̂, KV, S, f):
  KV′, π_f ← P.Init(KV, π̂)
  S_L1, S_L2, S_L3 ← Configure(S)
  δ ← Weights(S_L2, KV′)
  return KV′, (S_L1, S_L2, S_L3), δ

s_L1.ProcessQuery(k, v):
  ℓ ← P.Batch(k)
  For ((k, j), v) ∈ ℓ:
    s_L2 ← S_L2[H(k)]
    s_L2.Enqueue((k, j, v))

s_L2.Process():
  k, j, v ← Dequeue()
  v ← P.UpdateCache(k, j, v)
  s_L3 ← S_L3[H(F(k, j))]
  s_L3.Enqueue(s_L2, (F(k, j), v))

s_L3.Process(δ):
  s_L2 ←δ S_L2
  k′, v ← Dequeue(s_L2)
  v ← ReadThenWrite(KV′, k′, v)
  return k′, v
```

Figure 8: **SHORTSTACK initialization and processing logic at L1, L2 and L3 servers.** S_{L1}, S_{L2}, S_{L3} are the sets of proxy servers in each layer, and S is the set of physical servers upon which they are initialized. (k, v) corresponds to the plaintext key-value pair, while j is the replica identifier for a given replica of the key. F is a secretly keyed pseudorandom function and $\mathcal{H}$ is a consistent hash function.



(a) Round robin scheduling of queires at L3 queues results in non-uniform distribution

(b) Weighted scheduling at L3 queues ensures uniform distribution

Figure 9: Query-scheduling at L3 layer should ensure uniform distribution over ciphertext keys for security. In each figure, (left) shows the probability of scheduling queries from each of the L2 servers, while (right) shows the resulting distribution across ciphertext keys.

Finally, each L3 server maintains a separate queue for each L2 server it receives queries from, and dequeues queries from the queues following a biased distribution determined by the weight vector δ. To hide whether the query is a read or a write, SHORTSTACK employs the standard approach from prior oblivious data access schemes of performing each queries as a read followed by a write to the KV store. Specifically, in Figure 8's ReadThenWrite() method, the L3 server first reads and decrypts the value associated with the query from the key-value store. If the value needs to be updated (i.e., write query), then the plaintext value is updated accordingly. Finally, the L3 server writes the (re)encrypted value for the query back to the KV store.

Query scheduling at L3 layer for security. The way in which queries from different L2 servers are scheduled at each L3 server has security implications. As a concrete example, consider a scenario where three plaintext keys a, b and c with 6, 4 and 2 replicas (or, ciphertext keys), respectively, are mapped to three different L2 servers P_1, P_2 and P_3. Suppose we have two L3 servers, and one of these handles half of the ciphertext keys for each plaintext key (Figure 9 illustrates the example, focusing only on one of the L3 servers and the ciphertext keys mapped to this server). If the L3 server processes queries from each server with equal likelihood (*e.g.*, using round-robin scheduling), then the distribution across ciphertext keys would no longer be uniform, since queries from the first server would be under-sampled while those from the third server would be over-sampled (Figure 9 (a)). To ensure L3 servers still issue queries that are uniform random, they maintain a separate query queue for each L2 server, and process the queues in proportion to the volume of traffic the corresponding L2 servers generate. In the above example, the L3 server would schedule queries from each of the L2 servers with probabilities $3/6$, $2/6$ and $1/6$, respectively, leading to a uniform distribution across ciphertext keys (Figure 9 (b)).

Accurately estimating the access distribution. SHORTSTACK employs a lightweight mechanism through which a single L1 proxy server can observe all client queries, enabling distribution estimation as accurately as a centralized proxy system [6]. One of the L1 proxy servers, designated as the leader, monitors the access distribution (handling failures will be discussed in the next subsection). Upon receiving a query, an L1 proxy server asynchronously forwards the corresponding plaintext key—and not the entire query—to the L1 leader, ensuring that the leader has a complete view of the access distribution. Sending the plaintext key and not the entire query to the leader is an useful optimization for both read and write queries—it reduces the additional network load (for write queries, values are not forwarded; for read queries, the responses are not forwarded) since the plaintext key is typically much smaller than the value itself (*e.g.*, 8B keys for 1KB value in [41]). As such, this has negligible impact on SHORTSTACK scalability and performance.

4.3 Handling Failures

We now describe how SHORTSTACK ensures fault-tolerance while preserving security and correctness under failures. We assume the standard *fail-stop* failure model [36, 39]. SHORTSTACK employs a separate centralized *coordinator* node which keeps track of the health of the proxy servers using heartbeats, detects failures, and notifies other proxy servers as needed to designate a fail-over node. The coordinator node is also replicated using ZooKeeper [42] for strong consistency. As such, a $(2r+1)$-replicated coordinator can tolerate up to r failures without any security or performance consequences.

Handling L1 and L2 failures. Failure of a single L1 server does not impact the availability of SHORTSTACK, as future client queries could potentially be load balanced across the remaining L1 servers. Such a failure, however, has security implications—consider the case where an L1 proxy server fails in the middle of forwarding a batch of queries, i.e., some of the queries in the batch have been forwarded, but others are lost due to the failure. Any real queries that are lost would need to be retried by clients. The retried queries would now result in the same real accesses, but with *new* fake accesses generated. This permits an adversary to identify real queries with high confidence by simply isolating the repeated accesses due to failures. To protect against such a vulnerability,

SHORTSTACK ensures the following invariant:

Invariant 1 (Batch atomicity)**.** *Either all of the queries in a batch are forwarded to the KV store, or none of them are.*

SHORTSTACK achieves this by replicating the state of the L1 proxy servers across multiple replicas ($f+1$ replicas to tolerate up to f failures) using chain replication protocol [39]. As shown in Figure 7, SHORTSTACK maintains staggered chains across a fixed pool of physical servers, such that each physical server hosts the head node of a single chain. Chain replication ensures that all L1 replicas in the chain buffer a batch of queries, before the queries are forwarded to L2 servers—the buffered batches are only cleared when all corresponding acknowledgements are received from the L2 servers. As such, as long as any L1 replica in the chain is online, the set of buffered batches is available and can be used to retry queries as required, ensuring Invariant 1.

Since L2 servers store UpdateCache partitions, ensuring fault-tolerance for them is crucial for availability, correctness and security. As such, SHORTSTACK replicates the UpdateCache state for any key across multiple L2 proxy replicas using chain replication, similar to L1 servers.

Within each chain of L1 and L2 servers, failures of replicas are handled as per the standard chain replication protocol [39]. Since the L1 server chains interact with the L2 server chains, additional failure handling is necessary in certain cases. Consider the interaction between an L1 tail and an L2 head: if the L1 tail fails, its predecessor in the chain becomes the new tail, and resends the queries in the buffered (unacknowledged) batches to the corresponding L2 head replicas. The L2 servers, on the other hand, discard the queries that they have already seen and forward the remaining down the chain. SHORTSTACK facilitates the detection of duplicate queries by assigning unique sequence numbers to each query. If an L2 head fails, on the other hand, its successor become the new head. All L1 tails then examine their buffered batches to resend queries that were destined to the failed L2 head. As before, the new L2 head simply discards any queries that it has already seen, forwarding the remaining down the chain.

Handling L3 failures. Unlike L1 and L2 servers, L3 servers are not replicated, and hence entail different failure handling. Since L3 servers are stateless, if an L3 server fails, the remaining L3 servers can assume the responsibility of the ciphertext labels that the failed server was handling. Since the system remains available as long as at least one of the L3 servers is online, we need at least $f+1$ L3 servers to tolerate f failures. However, there are two subtle issues that can arise due to L3 failures—we describe these next, along with how SHORTSTACK addresses them.

On an L3 server failure, queries that were in-flight at the failed L3 server would be lost, which can then be retried by the L2 servers. Note that such retries can cause duplicate queries being sent to the KV store. Since the duplicate queries are to uniformly accessed ciphertext keys, it may seem like they do not reveal any distribution-sensitive information. However, repeating the queries in exactly the same order (or a correlated order) introduces a subtle security vulnerability. Specifically, when an L3 server fails, L2 tail servers repeat buffered queries (which are uniform random) and redistribute them to different L3 servers. If the order of these queries is exactly the same as before, an adversary can identify the sequences of repeated queries and correlate them to the L2 server that generated those queries. Moreover, the adversary can also map the specific ciphertext keys corresponding to the plaintext keys managed by a particular L2 server, revealing distribution sensitive information. To prevent this leakage, SHORTSTACK *randomly shuffles* buffered queries before repeating them—we formally prove in [43] how this ensures security under L3 server failures.

Recall that the L3 server performs a read followed by a write for all queries. For read queries (fake or real), the write simply writes back the value read from the KV store, i.e., a *fake* write. This can lead to consistency issues during failure of L3 servers—fake in-flight write queries sent by a failed L3 server prior to failure could be delayed by the network and overwrite a real write query sent by the new L3 proxy server responsible for the same ciphertext key. To address this issue, after an L3 failure, the L2 servers delay repeating buffered queries for a fixed amount of time to allow potential in-flight queries from the failed L3 server to get delivered to the KV store. We select the wait time at L2 servers long enough to ensure *all* in-flight queries are propagated to the KV store.

4.4 Handling Dynamic Distributions

Designing distributed, fault-tolerant, oblivious data access systems is challenging when underlying distribution can change over time. We outline two reasons. First, the centralized proxy design (§2.2) relies on having a complete view of the underlying distribution to detect and to react to distribution changes. Detecting the change when queries are spread across multiple proxy servers, and informing other proxy servers about the same, introduces the first challenge. Second, if different proxy servers independently initiate and terminate the replica swapping phase at different times, the resulting distribution may not appear uniform random to an adversary. As such, the adversary may be able to leverage this information to identify the keys that may have changed in popularity. We next discuss how SHORTSTACK resolves these challenges.

To detect distribution changes, SHORTSTACK leverages the L1 leader, which has visibility of all client queries (§4.2). The L1 leader is responsible for monitoring the access distribution and employs standard statistical tests to check if there is a change in distribution (i.e., from $\hat{\pi}$ to $\hat{\pi}'$) similar to PANCAKE. Upon detecting a change in distribution, the L1 leader initiates the distribution change process. To ensure security and correctness during distribution change, the L1 leader employs a specialized protocol inspired by two-phase commit (2PC) [44] to facilitate an *atomic* transition from $\hat{\pi}$ to $\hat{\pi}'$ across all servers

in its three-layer design, both during the initiation and termination of the replica-swapping phase employed by PANCAKE. Our 2PC-based approach guarantees:

Invariant 2 (Distribution change atomicity)**.** *Once any L3 proxy server issues a query according to $\hat{\pi}'$, all subsequent queries issued by any L3 server must be according to $\hat{\pi}'$.*

In other words, there is an instant of time t_c in the protocol's execution, such that: (1) before t_c, all queries are processed according to the distribution $\hat{\pi}$, and (2) after t_c, all queries are processed using $\hat{\pi}'$. This allows us to ensure security for SHORTSTACK even under dynamic distributions, as we detail in §5. The invariant also ensures consistency during distribution change. In particular, since the change of distribution can result in a change in number of replicas for various plaintext keys, the invariant ensures queries from old and new distributions are not mixed together; this guarantees consistency by ensuring stale replicas from the old distribution are not updated incorrectly due to the new distribution by different L2 proxy servers. We show that our protocol guarantees the above invariant, with a precise specification in [43]. Failures during the above protocol are handled transparently by chain replication as L1, L2 servers are chain replicated. This ensures that even with failures during protocol execution, Invariant 2 is still preserved. As demonstrated in §6, SHORTSTACK can recover from such failures quickly enough so as to ensure that their effects are not perceptible to an adversary.

5 Security Analysis

This section presents a security model for access pattern attacks on a system with distributed, *fault-tolerant* proxy servers, and a proof that SHORTSTACK achieves security under this model.

5.1 Need for New Security Definitions

State-of-the-art ROR (real-or-random indistinguishability) based security definitions for access pattern attacks [6] are unable to capture the security implications of our distributed proxy setting due to two main reasons. First, ROR-based definitions focus on indistinguishability between a real and a uniform random distribution (over the entire support). However, as discussed in §3.2, we do not yet know whether it is possible to guarantee uniform random distribution over the entire support during failures for *any* distributed proxy architecture. Our IND-based security model and definitions capture the powerful intuition that uniform random distribution is not even necessary: even though the distribution is non-uniform under failures, the adversary does not gain any *usable* advantage as long as the final distribution is independent of the real distribution. More precisely, our IND-based security focuses on demonstrating indistinguishability between two arbitrary input distributions. As we will show, under our model, the only information revealed to the adversary is that a failure occurred, information the adversary already possess; it cannot, however, use this information in inferring any information about the underlying distribution itself. While it is not uncommon for IND security to reduce to ROR security in many settings, this is clearly not the case in our setting if (and, as we note later, only if) there are failures.

$\text{IND-CDFA}^{\mathbb{A}}_{b,q,S,f,\pi_0,\hat{\pi}_0,\pi_1,\hat{\pi}_1}$:
$\mathsf{KV}, \mathcal{T}, \mathsf{st}_A \leftarrow\$ \mathbb{A}_1(f, S)$
$(\mathsf{KV}', \mathcal{C}, \delta) \leftarrow\$ \mathsf{Init}(\hat{\pi}_b, \mathsf{KV}, S, f)$
For i in 1 to q:
  $w \leftarrow\$ \pi_b$
  $W \leftarrow W \cup \{w\}$
$\tau_1, \tau_2, \ldots \leftarrow \mathsf{Process}(W, \mathcal{C}, \mathcal{T}, \mathsf{KV}', \delta)$
$b' \leftarrow\$ \mathbb{A}_3(\mathsf{st}_A, \mathsf{KV}', \tau_1, \tau_2, \ldots)$
return b'

Figure 10: **IND-CDFA security game.**

The second reason for needing new security model and definitions is that ROR-based definitions fail to capture the impact of query reordering on the transcripts observed by an adversary due to (i) distributed query processing, and (ii) worst-case timings of proxy failures. Specifically, a key challenge in demonstrating security lies in precisely capturing the effect of the distributed and failure-prone execution of any scheme in a sequential game-based proof, which the ROR-based approach omits. We thus have to develop accurate *simulators* that transform distributed query processing to an equivalent sequential one. Our model and definitions are not specific to SHORTSTACK, and can be used as templates for any distributed, fault-tolerant, proxy design.

When there are no failures, our security definition captures the same security guarantees as prior work [6] — our extensions to the model are required to capture the effect of failures in the distributed proxy setting. In incorporating these extensions, we have only strengthened the adversary.

5.2 Security Definitions and Proof of Security

We call our security definition Indistinguishability under Chosen Distribution and Failure Attack, or IND-CDFA (Figure 10). The game IND-CDFA is parameterized by bit b (to pick one out of the two given distributions), number of queries q, the set of proxy servers S on which the distributed oblivious data access protocol runs, the maximum number of server failures f allowed (similar to classical distributed systems literature that provides fault tolerance up to a fixed number of failures), and two distributions (and their estimates) that the adversary tries to distinguish between.

The adversary first outputs KV pairs KV and a queue $\mathcal{T}$ of at most f failure events. Each failure event e is characterized by the tuple (n, t, γ, r), where n is the server in S that fails, t is the time at which the last query is issued by n before failure, $t - \gamma$ is the time at which the last query was acknowledged at n before failure, and r is the failure recovery time. Next, the distributed proxy scheme's Init function generates transformed KV pairs KV', a set of (potentially replicated) servers $\mathcal{C}$, and internal state δ specific to the scheme. For instance, in SHORTSTACK, $\mathcal{C}$ consists of two sets of replicated server

chains (with replication factor $f+1$) corresponding to L1 and L2 layers, and a set of $> f$ unreplicated servers for the L3 layer. The state δ corresponds to weights for L3 servers used in query scheduling, as outlined in §4.2.

After initialization, q queries are drawn from the distribution π_b and populated into the vector W. The proxy scheme's Process function takes W, $\mathcal{C}$, $\mathcal{T}$, KV′ and δ as input, and generates the output transcripts τ, which is fed to the adversary to try and guess the underlying distribution (i.e., the bit b). The adversary "wins" if it guesses b correctly. Intuitively, the security goal captured by the definition rules out access pattern attacks since the probability of accessing an encrypted label in KV′ is independent of the underlying distribution itself, and an adversary cannot determine which distribution was used to generate accesses to KV′.

Note that, IND-CDFA definition is independent of SHORTSTACK's design. Specifically, our definitions only assume the presence of multiple failure-prone proxy servers which are initialized using an Init function and process queries using a Process function, neither of which are specific to SHORTSTACK. Thus, our security model and definitions can be used to study oblivious data access properties of any distributed system that can factor its initialization and query processing logic along these two functions.

The following theorem establishes the security of SHORTSTACK under IND-CDDFA:

Theorem 1 (IND-CDFA Security)**.** *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi_0, \hat{\pi}_0, \pi_1, \hat{\pi}_1$ be query distributions. For any q-query* IND-CDFA *adversary $\mathbb{A}$ against* SHORTSTACK *there exist adversaries $\mathbb{B}$, $\mathbb{C}$, $\mathbb{D}_1$, $\mathbb{D}_2$ such that*

$$\mathbf{Adv}^{ind\text{-}cdfa}_{\text{SHORTSTACK}}[(\mathbb{A})] \leq \mathbf{Adv}^{prf}_{F}[(\mathbb{B})] + \mathbf{Adv}^{ror}_{E}[(\mathbb{C})] + \mathbf{Adv}^{dist}_{Q,\pi_0,\hat{\pi}_0}[(\mathbb{D}_1)] + \mathbf{Adv}^{dist}_{Q,\pi_1,\hat{\pi}_1}[(\mathbb{D}_2)]$$

where F, E are PRF, AE schemes used by SHORTSTACK*. Adversaries $\mathbb{B}, \mathbb{C}, \mathbb{D}_1, \mathbb{D}_2$ run in same time as $\mathbb{A}$ with Q queries.*

Our security proof stems from three key components:

- Security of E as a randomized authentication scheme applied over values and F as a pseudorandom function applied over keys; this is rigorously analyzed in prior work [45, 46].
- Our estimate $\hat{\pi}$ of the underlying distribution π is sufficiently accurate. While this estimate may not be perfect, our security model only requires that $\hat{\pi}$ and π be indistinguishable for a limited number of samples, which holds for estimators used in prior work [6] on real-world workloads [41]. Since our design employs a single leader L1 server to estimate the underlying distribution using the keys for all client queries (§4.2) and employs the same estimators as prior work, its estimation is just as accurate.
- Accesses issued to the KV store reveal nothing about the underlying distribution π.

To prove the third component, we introduce simulators to sequentialize the distributed execution of SHORTSTACK's query processing to make it compatible with our game-based definition (Figure 10). Specifically, we simulate Process function for SHORTSTACK by first generating the intermediate transcript, β, assuming no failures. We do so by (i) going layer by layer and executing processing logic at appropriate servers in SHORTSTACK, and (ii) incorporating the impact of network reordering across queries between layers. We then use a Transform simulator to capture the effect of failures and generate the final transcripts τ from β. We do so by recursively applying the effect of L3 server failure events in $\mathcal{T}$ on the intermediate transcripts β in the order that they occur.

Finally, we show that the final transcripts τ are independent of intermediate transcripts β, and then show that β are independent of the underlying distribution π. The first part holds since SHORTSTACK randomly shuffles buffered queries before replaying them post failure (§4.3) and failure recovery time in SHORTSTACK is short enough to not be visible to an external observer given our failure model (§4.3) and as shown empirically in (§6.2). The second part holds, since the underlying oblivious data access scheme [6] in SHORTSTACK generates uniform random queries (§4.2) and network reorderings between layers are independent of π.

Finally, to model dynamic distributions, we generalize the above definition to Indistinguishability under Chosen Dynamic Distribution and Failure Attack or IND-CDDFA. This definition, along with the proof of SHORTSTACK's security under it, formal descriptions of our simulators, and the proof for independence of τ and π are presented in [43].

6 Evaluation

SHORTSTACK is implemented in $\sim$ 6k lines of C++, using Thrift as the RPC library, AES-CBC-256 for encrypting values, HMAC-SHA-256 as our PRF, and Redis as the KV store.

Compared systems. We compare SHORTSTACK performance against two baselines. The first baseline is distributed, but encryption-only, that is, it encrypts data and client queries, but does not guarantee oblivious data access; here, client queries are randomly load balanced across stateless proxy servers that perform encryption/decryption and forward queries to the KV store. This baseline serves as an upper bound on the performance that can be achieved by any oblivious data access system (including SHORTSTACK). The second baseline is a centralized PANCAKE [6] proxy server. While this suffers from security and availability problems in the face of failures (§3.1), its performance serves as a reference point for understanding SHORTSTACK's scalability.

Experimental setup. We run our experiments on Amazon EC2. By default, we host the proxy instances across c5.4xlarge VMs with 16 vCPUs (8 cores with 2 threads per core), 32 GB RAM and 10Gbps network links. In order to emulate a cloud KV store with practically infinite bandwidth,

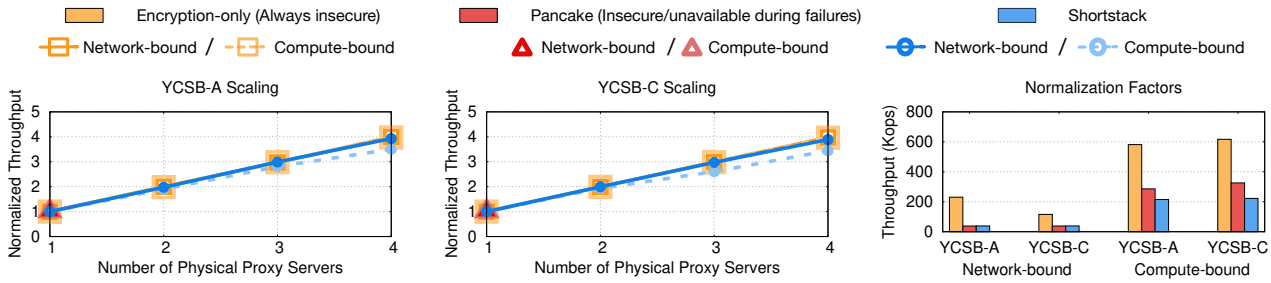


Figure 11: **Scalability properties of different systems when network bandwidth and compute are the bottleneck.** (left, middle) show system throughput normalized by throughput for a single physical proxy server, while (right) shows system throughput with a single physical proxy server. The Encryption-only lines for the network-bound and compute-bound cases overlap, since its throughput scales linearly in both cases. Since Pancake is centralized, it only has a single data point at $X = 1$ for each of the cases, and these points overlap. See §6.1 for details.

we use a single powerful VM, c5d.metal (96 vCPUs, 128 GB RAM) with large network bandwidth (25 Gbps). Similar to prior work [6], we emulate WAN access link bandwidth by throttling the bandwidth from each proxy server to the KV store server to 1Gbps. The clients run on lightweight t3.2xlarge VMs (8 vCPUs, 32GB RAM) in the same LAN. Both PANCAKE and SHORTSTACK use a batch size of $B = 3$.

Dataset and Workloads. We use the standard YCSB benchmark [41] to generate our dataset and workloads. The dataset comprises 1 million KV pairs, with 8B keys and 1KB values. We use workloads A (50% reads, 50% writes) and C (100% reads) for our experiments. YCSB workloads perform accesses distributed according to the Zipfian distribution [41]; unless otherwise stated, the skewness parameter for the Zipfian distribution in our experiments is set to the YCSB default of 0.99 (that is, heavily skewed), which is representative of many real world workloads. We also perform sensitivity analysis against distribution skew.

6.1 Scalability Analysis

We now analyze SHORTSTACK's scalability with varying number of physical proxy servers under different workloads.

Throughput scaling under bandwidth bottleneck. We study throughput scaling for SHORTSTACK by varying the number of physical proxy servers and comparing its performance against the baselines. For SHORTSTACK, k physical proxy servers constitute k chain-replicated L1 instances with $\min(k,3)$ replicas each, k chain-replicated L2 instances with $\min(k,3)$ replicas each, and k unreplicated L3 instances (i.e., the system can tolerate up to $\min(k,3)-1$ failures). For the encryption-only baseline, a separate proxy instance is run on each physical proxy server, and the PANCAKE baseline always uses only one physical proxy server.

Figure 11 shows the scalability results for two cases: one where the physical proxy servers are network-bound (solid lines), and another where they are compute-bound (broken lines). We begin with the former case; we see that SHORTSTACK throughput scales linearly with the number of physical proxy servers. Note that we normalize each system's throughput by its throughput with a single physical proxy server — Figure 11 (right) shows normalization factors for each system, i.e., throughput with single physical proxy server. The red cross shows the throughput of the PANCAKE baseline (38 KOps): SHORTSTACK's distributed design enables linear throughput gains relative to PANCAKE via scaling. The insecure baseline also scales linearly due to random load-balancing across its proxy instances. Since all proxy servers are network bound, SHORTSTACK incurs only a constant overhead (corresponding to the relative bandwidth increase due to the oblivious data access protocol) compared to the encryption-only baseline for all configurations as we scale the number of physical proxy servers. For the YCSB-C workload, the gap between SHORTSTACK and Encryption-only baseline throughput stems from the 3× overhead imposed by the PANCAKE protocol for a batch size of $B = 3$. For the YCSB-A workload, however, the encryption-only baseline throughput is 6× higher than SHORTSTACK since it can exploit the bidirectional bandwidth to the KV store for 50% reads and 50% writes. SHORTSTACK, however, already issues a read followed by a write for every query, so it is unable to similarly exploit the bidirectional bandwidth. Since YCSB-A has equal proportion of read and write queries, this situation corresponds to the worst-case bandwidth increase (6×) for SHORTSTACK relative to the encryption-only baseline.

Throughput scaling under compute bottleneck. We now analyze throughput scaling when the physical proxy servers are compute-bound: we re-run the same experiments as above, but using c5.metal EC2 VMs (96 vCPUs, 192GB RAM, 25Gbps network bandwidth) for all systems *without* throttling the access link bandwidth to the KV store server. As the broken lines corresponding to the compute-bound case in Figure 11 show, with a single physical proxy server SHORTSTACK achieves slightly lower throughput than PANCAKE for both workloads. This is because, under a compute bottleneck, SHORTSTACK incurs additional RPC processing overheads for communication between its layers. SHORTSTACK's throughput increases significantly with more physical proxy servers, achieving $3.4-3.6\times$ higher throughput with 4 physical proxy servers. The increase in throughput is not perfectly linear, since workload skew results in load imbalance at the L2 layer. This effect

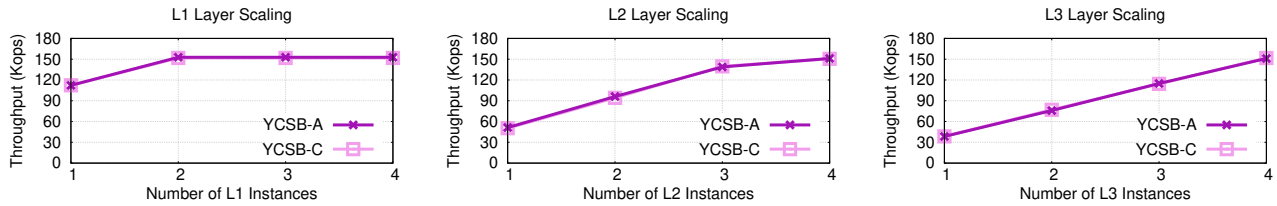
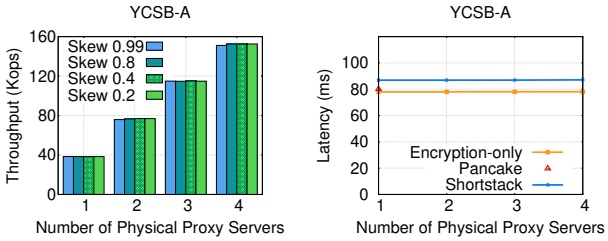


Figure 12: **SHORTSTACK layer-wise scaling for YCSB workloads A and C.** See §6.1 for details.



(a) SHORTSTACK throughput is unaffected by access skew.

(b) Query latency vs. number of physical proxy servers.

Figure 13: **SHORTSTACK throughput scaling with varying skew (a) and SHORTSTACK latency overheads (b).** See §6.1 for details.

is not observed for the network-bound case, since the network bandwidth between L3 instances and the KV store is bottlenecked before workload skew causes compute at the L2 layer to become bottlenecked. For the reminder of our evaluation, we use the network-bound setting as our default configuration.

Understanding per-layer scalability bottlenecks. Our experiments in Figure 11 scale up all layers of SHORTSTACK in equal proportions as the number of physical proxy servers are increased. To better understand SHORTSTACK's bottlenecks, we now study scalability on a per-layer basis. Since each layer performs a different component of PANCAKE logic (§4), varying the scale of each layer independently while keeping the scale of the other two layers fixed allows us to understand which step becomes a throughput bottleneck before the others. For this, we use a setup similar to Figure 11. To understand L1 layer scalability, we fix the number of physical proxy servers to 4, the number of replicated L2 instances and unreplicated L3 instances to the default (4), and vary the number of replicated L1 instances from $1-4$. We perform similar experiments for the L2 and L3 layers as well. Figure 12 shows the corresponding results for the YCSB-A and YCSB-C workloads. For the L1 layer, throughput increases slightly from $X = 1$ to 2, beyond which it saturates, since L1 is no longer the bottleneck. For the L2 layer, from $X = 1$ to 3 throughput increases, albeit non-linearly due to plaintext key-based partitioning — while the number of plaintext keys handled by each L2 server is roughly equal, the number of replicas handled by them is skewed due to the skew in the YCSB workload. At $X = 4$, the L2 layer is no longer the bottleneck. For the L3 layer, throughput scales linearly from $X = 1$ to $X = 4$ due to ciphertext key-based partitioning, with each L3 proxy handling roughly the same number of ciphertext keys.

As expected, the bottlenecks are different at different SHORTSTACK layers. When all layers are sufficiently provisioned, SHORTSTACK is able to saturate the access link bandwidth between the L3 layer and the KV store. Reducing the number of L1 and L2 proxy instances, however, leads to *compute* becoming the bottleneck at the respective layers. One of the key contributors of compute overheads are serialization/deserialization for network queries. Finally, layer-wise scaling characteristics are similar for YCSB-C and YCSB-A workloads, as UpdateCache processing in YCSB-A due to writes does not account for much of the compute overheads.

Throughput scaling with skew. We evaluate SHORTSTACK scaling for workloads with different skew for a setup similar to Figure 11. We vary the skew parameter for YCSB's Zipf distribution from 0.2 (close to uniform) to 0.99 (heavy skew) to consider both extremes. We only show our results for YCSB-A in Figure 13(a), since results for YCSB-C were similar. SHORTSTACK system throughput scales linearly regardless of skew, because the bottleneck in the end-to-end query execution is the access link bandwidth between the L3 layer and the KV store for all scales. Since the skew only affects processing at L2 layer (which is not the bottleneck), our throughput is independent of skew. While SHORTSTACK throughput scales linearly even for heavily skewed workloads, there could indeed be rare extreme-case scenarios where such would not be the case, *e.g.*, if all popular plaintext keys get consistently hashed to a single L2 instance, resulting in a compute bottleneck at that instance.

SHORTSTACK Latency overheads. To quantify SHORTSTACK's latency overheads, we evaluate end-to-end query latency for varying number of physical proxy servers for compared systems using a setup similar to Figure 11 with one change: we separate the KV store and physical proxy servers by the WAN. Figure 13(b) shows the results; again, we only show YCSB-A workload results, as YCSB-C results are similar. Independent of the scale, SHORTSTACK increases query latency by a modest 8% (additional 6.8ms) compared to PANCAKE. This increase in latency is due to additional processing and network hops introduced by SHORTSTACK's multiple layers and chain replication within the L1 and L2 layers. Nevertheless, these overheads are masked by the significantly larger WAN access latency.

6.2 Failure Recovery

We now evaluate SHORTSTACK's ability to recover from failures and also validate our assumptions in proving SHORTSTACK security. We fix the number of physical proxy servers

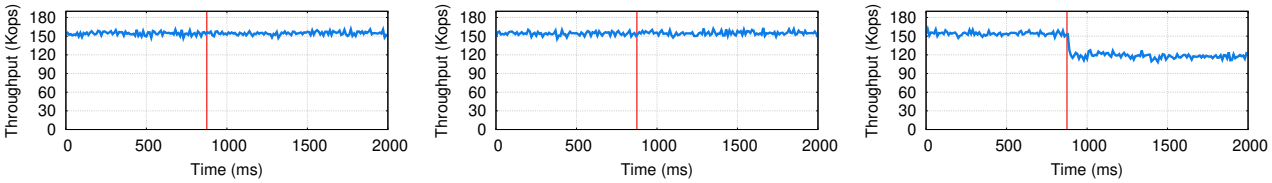


Figure 14: **SHORTSTACK failure recovery for (left) L1, (middle) L2, and (right) L3 failures.** See §6.2 for details.

to 4, the number of 3×-replicated L1, 3×-replicated L2, and unreplicated L3 instances to 4 each, and use the YCSB-A workload. To understand the impact of failures on each layer independently, we fail one proxy instance in a particular layer by killing its associated process; for L1 and L2, we kill an arbitrary replica from one of the instances. We measure the instantaneous throughput of our system during each experiment at 10ms granularity; when measured at finer-grained timescales, we found that the instantaneous throughput numbers were too noisy to discern any meaningful trends.

Figure 14 shows the effect of failure at each layer on SHORTSTACK throughput. We find that failures in L1 and L2 proxy chains do not cause any noticeable dip in the throughput, since SHORTSTACK can quickly recover from failures within 3–4 ms — much faster than the average query latency over WAN (∼ 90ms), and smaller than the typical variance in query latencies. Hence, an adversary cannot reliably distinguish between a failure event and variations in instantaneous throughput due to noise caused by network delays, independent of the timescale at which measurements are done. This validates our assumption for SHORTSTACK security under failures discussed in §5 — specifically, L1 and L2 failures have an imperceptible impact on an adversary's observed access pattern to the KV store. Upon an L3 proxy failure, the throughput reduces by 25% – commensurate with the reduction in the bandwidth to the KV store server; however, since L3 layer partitions queries by ciphertext keys, it does not reveal any information about the client access patterns.

7 Related Work

We now discuss the works most closely related to SHORTSTACK's goals of distributed, fault-tolerant, oblivious data access. ORAM [10] approaches have been adapted to real world cloud storage [12–17, 26], with recent efforts enabling *concurrency* and *asynchrony*. Oblivious Parallel RAM (OPRAM) [12, 14, 47–50] permits multiple concurrent clients to query the storage, but requires cross-client coordination per-query (*e.g.*, using oblivious aggregation [12]) to ensure no two clients concurrently issue a query for the same data. This severely limits throughput scaling under high query traffic due to compute bottlenecks.

CURIOUS [16] and TaoStore [15] employ a centralized proxy model, but permit client parallelism via *asynchrony*. Since each operation requires updates to per-plaintext key proxy state for multiple random KV pairs, extending their design to a distributed and secure one is challenging. The latest in this line of work, ConcurORAM [17] and Snoopy [26], permit multiple parallel clients to query a cloud-hosted ORAM *without* inter-client or proxy based coordination. ConcurORAM achieves this by offloading much of the synchronization to the cloud, which not only requires non-trivial changes to cloud storage, but also limits system throughput under high load. Concurrent to our work, Snoopy builds a distributed oblivious data access system (for ORAM-based designs); however, Snoopy does not prove security for scenarios where servers can fail. In any case, SHORTSTACK and Snoopy offer the same trade-offs as discussed in [6]—Snoopy can handle active adversaries, but also incurs significantly higher overheads relative to SHORTSTACK. Prior work [6] has empirically shown that state-of-the-art single proxy ORAM schemes achieve 220× lower throughput than PANCAKE for the same workloads as in our evaluation. Since SHORTSTACK can scale PANCAKE's throughput linearly (§6) with number of proxy servers, even if one could design a distributed ORAM system that scales near-perfectly with number of proxy servers, the maximum achievable throughput would still be orders of magnitude lower than SHORTSTACK.

8 Conclusion

Existing systems for oblivious data access rely on a centralized, stateful, proxy to coordinate queries between applications and the storage server. We have demonstrated that, in failure-prone deployment, such systems can suffer from security violations, long periods of unavailability and/or scalability limits. Our core contribution is SHORTSTACK, a distributed, fault-tolerant and scalable system for oblivious data access. Using a novel layered architecture, SHORTSTACK achieves the classical obliviousness guarantee—access patterns observed by the adversary being independent of the input—even under a powerful passive persistent adversary that can force failure of arbitrary (bounded-sized) subset of proxy servers at arbitrary times. We also introduce a security model to study oblivious data access with distributed, failure-prone, servers.

Acknowledgements

We would like to thank our shepherd, Alex C. Snoeren, and the anonymous OSDI reviewers for their insightful feedback. We would also like to thank Thomas Ristenpart for many useful discussions during this work. This research was supported in part by NSF awards 2054957, 2047220, 2118851, 1704742, Faculty Research Awards from Google and NetApp, and an IC3 fellowship thanks to IC3 industry partners.

References

[1] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[2] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[3] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[4] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE S&P*, 2019.

[5] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *IEEE S&P*, 2020.

[6] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*, 2020.

[7] Care Cloud. 5 advantages of a cloud-based EHR. https://www.carecloud.com/continuum/5-advantages-of-a-cloud-based-ehr-for-small-practices/.

[8] Alex Mu-Hsing Kuo. Opportunities and challenges of cloud computing to improve health care services. *JMIR*, 2011.

[9] Microsoft. Healthcare-europe. https://www.microsoft.com/en-ie/lcc_cloud/healthcare-europe.

[10] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *JACM*, 1996.

[11] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, 2018.

[12] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In *CCS*, 2012.

[13] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious cloud storage. In *IEEE S&P*, 2013.

[14] Jacob R. Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *FAST*, 2013.

[15] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *IEEE S&P*, 2016.

[16] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In *CCS*, 2015.

[17] Anrin Chakraborti and Radu Sion. ConcurORAM: High-throughput stateless parallel multi-client ORAM. In *NDSS*, 2019.

[18] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In *SIGMOD*, 2015.

[19] Marie-Sarah Lacharite and Kenneth G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology*, 2018.

[20] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ITCS*, 2016.

[21] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *CRYPTO*, 2018.

[22] Giuseppe Persiano and Kevin Yeo. Lower bounds for differentially private rams. In *EUROCRYPT*, 2019.

[23] Mor Weiss and Daniel Wichs. Is there an oblivious RAM lower bound for online reads? In *TCC*, 2018.

[24] Kasper Green Larsen, Mark Simkin, and Kevin Yeo. Lower bounds for multi-server oblivious rams. In *TCC*, 2020.

[25] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. What storage access privacy is achievable with small overhead? In *PODS*, 2019.

[26] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.

[27] Securing cloud services for health. https://www.enisa.europa.eu/news/enisa-news/securing-cloud-services-for-health.

[28] French decision to have microsoft host health data hub still attracts criticism. https://www.euractiv.com/section/health-consumers/news/french-decision-to-have-microsoft-host-health-data-hub-still-attracts-criticism/.

[29] Microsoft cloud services will store and process eu data within the eu. https://www.privacy-ticker.com/microsoft-cloud-services-will-store-and-process-eu-data-within-the-eu/.

[30] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[31] Baffle. https://baffle.io.

[32] Ciphercloud. http://www.ciphercloud.com/.

[33] Navajo Systems. http://tinyurl.com/y85obds6.

[34] Perspecsys: A Blue Coat Company. http://perspecsys.com.

[35] Skyhigh Networks. http://www.skyhighnetworks.com.

[36] Richard D Schlichting and Fred B Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *TOCS*, 1983.

[37] Zhao Chang, Dong Xie, and Feifei Li. Oblivious ram: A dissection and experimental evaluation. In *VLDB*, 2016.

[38] Original buttermilk pancakes - (short stack). https://www.ihop.com/en/menu/world-famous-buttermilk-pancakes-and-crepes/original-buttermilk-pancakes-short-stack.

[39] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[40] Scott Lystig Fritchie. Chain replication in theory and in practice. In *Erlang*, 2010.

[41] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[42] Apache zookeeper. https://zookeeper.apache.org/.

[43] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. Shortstack: Distributed, fault-tolerant, oblivious data access. Cryptology ePrint Archive, 2022. https://eprint.iacr.org/2022/662.

[44] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. 1987.

[45] Oded Goldreich, Shaffi Goldwasser, and Silvio Micali. How to construct random functions. *JACM*, 1986.

[46] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In *EUROCRYPT*, 2006.

[47] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram and applications. In *TCC*, 2016.

[48] T-H Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel ram. In *TCC*, 2018.

[49] T-H Hubert Chan and Elaine Shi. Circuit opram: Unifying statistically and computationally secure orams and oprams. In *TCC*, 2017.

[50] Gareth T Davies, Christian Janson, and Daniel P Martin. Client-oblivious opram. In *ICICS*, 2020.

Groove: Flexible Metadata-Private Messaging

Ludovic Barman
EPFL

Moshe Kol
Hebrew University of Jerusalem

David Lazar
EPFL

Yossi Gilad
Hebrew University of Jerusalem

Nickolai Zeldovich
MIT CSAIL

Abstract

Metadata-private messaging designs that scale to support millions of users are *rigid*: they limit users to a single device that is online all the time and transmits on short regular intervals, and require users to choose precisely when each of their buddies can message them. These requirements induce high network and energy costs for the clients, restricting users to communicate via one powerful device, like their desktop.

Groove is the first scalable metadata-private messaging system that gives users *flexibility*: it supports users with multiple devices, allows them to message buddies at any time, even when those buddies are offline, and conserves the user's device bandwidth and energy. Groove offers flexibility by introducing *oblivious delegation*, where users designate an untrusted service provider to participate in rigid mechanisms of metadata-private communication. It provides differential privacy guarantees on par with rigid systems like Stadium and Karaoke.

An evaluation of a Groove prototype on AWS with 100 servers, distributed across four data centers on two continents, demonstrates that it can achieve 32 s of latency for 1 million users with 50 buddies in their contact lists. Experiments with a client running on a Pixel 4 smartphone show that it uses about 100 MB/month of bandwidth and increases battery consumption by 50 mW (+16%) compared to an idle smartphone. These measurements show that Groove makes it realistic to hide messaging metadata on a mobile device.

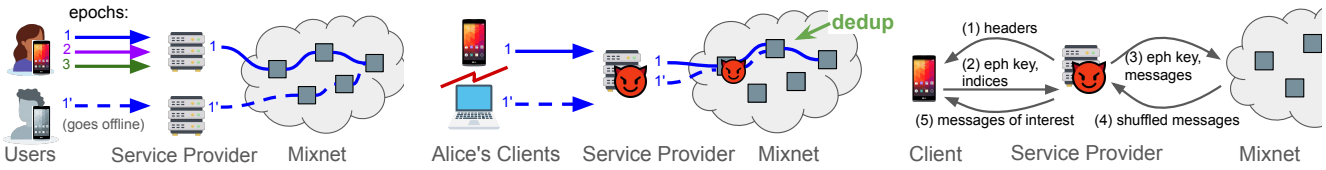
1 Introduction

There has been significant recent progress in scalable metadata-private messaging systems. These systems hide who communicates with whom and can support more users by deploying proportionally more servers (supporting a large user base is crucial for privacy [9]). However, systems that are scalable and provide strong privacy guarantees impose *rigid* requirements on users [16, 19, 20, 26, 27], like needing users to synchronize messaging into rounds and coordinate precisely in which rounds they will communicate with their buddy. If two buddies are not simultaneously online, they cannot communicate. Naively storing a message for a buddy to fetch later exposes the age of the message, which can be used to correlate the sender. Instead, this line of work has relied on expensive dialing protocols [21, 27] to coordinate conversations, which are impractical for mobile devices.

Rigidity in private messaging systems is inherently costly for clients. Since communicating users must be simultaneously online, an attacker monitoring the network can correlate buddies over time. To combat such attacks, clients submit and poll for messages at every round, leading to high bandwidth and energy overhead. This makes running a client on a phone prohibitively expensive and effectively requires the users to have an always-on desktop computer (e.g., at home). However, many users do not have such a computer, e.g., if they use a laptop that they carry with them. Finally, using metadata-private messengers from multiple devices poses a risk if the devices become partitioned and accidentally send multiple message to the same buddy in the same round (which reveals the sender). These limitations stand out compared to traditional messaging systems (without metadata privacy) and hinder adoption.

We present Groove, the first *flexible* messaging system that provides metadata privacy and scales well to support a large user base. Groove's users can send messages to any of their buddies from any device, and can go offline and retrieve messages sent to them later. Groove provides similar messaging latency to recent metadata-private communication systems under the same global active adversary model [15, 16, 19, 20, 26]. Groove builds on mixnets, where servers shuffle messages in batches, to unlink senders from their messages. However, mixnets are inherently rigid: they require all users to submit a message in every round to mix all conversations together, and for all users to receive messages from the mixnet at the same rate to avoid correlated traffic patterns between buddies. To handle this rigidity, Groove introduces *oblivious delegation* to an untrusted service provider. Groove clients interact with a service provider who participates in the rigid mixnet protocol on their behalf and synchronizes multiple devices. Oblivious delegation ensures that an adversary can not learn anything about users' communication metadata by compromising their service providers, even if the users' devices go offline or get partitioned. Achieving oblivious delegation involves three mechanisms, as follows.

Non-interactive setup (Figure 1a). Establishing a message channel between buddies requires them to submit just one setup message through the mixnet, without waiting for their peer's response. A non-interactive setup protocol is crucial since buddies might not be simultaneously online to run the



(a) *Non-interactive Setup*. Groove's async setup protocol enables metadata-private messaging between buddies, even if they are not simultaneously online. The service provider buffers setup messages and submits them to the mixnet at the right time.

(b) *Uncoordinated Replacement*. Groove allows multiple clients (on different devices) of the same user to interact with the system safely, even if they cannot coordinate. Clients can replace old messages buffered at the service provider to support new buddies.

(c) *Oblivious Fetch*. Groove's clients avoid retrieving cover traffic messages stored at the service provider, without revealing which messages they fetch. The servers only process a user's messages once per client, regardless of the number of messages stored.

Figure 1: The three mechanisms that allow oblivious delegation in Groove.

protocol. Groove hides communication metadata even if only one buddy attempts this setup (or their peer's provider discards the other setup message). Message channels persist for long epochs, and this protocol allows Alice to prepare setup messages at her service provider well in advance and for many epochs. Service providers, if they're honest, submit setup messages to the mixnet at the right time, allowing Alice's buddies to send her messages even she goes offline. If the service provider misbehaves, messages might be (noticeably) lost but privacy is preserved, and Alice can switch to a different provider.

Uncoordinated replacement (Figure 1b). The adversary may partition a user's devices and prevent them from deciding which device communicates at a given round. If several devices of the same user accidentally submit messages to the same buddy in the same round, they can expose the recipient, who receives extra messages. Groove solves the issue with a new message replacement technique: each device can independently refresh the messages queued at the service provider without coordinating with other devices. Crucially, the protocol ensures that no metadata is revealed to the adversary, even if the provider is rogue and submits all messages (old and new) to the mixnet. This is achieved through path selection and message-tagging mechanisms, which ensure that each buddy receives at most one message from all of the user's devices. The replacement protocol allows clients to update setup messages, and hence to add buddies after preparing message channels for future epochs. It also allows users to switch between their devices, like traditional messaging applications.

Efficient messaging with many buddies (Figure 1c). Groove avoids expensive dialing (as in [21, 27]) by keeping a message channel open for each of a user's buddies. To minimize client costs when having many message channels, Groove's clients use a submission protocol that avoids sending a cover message for each idle channel, and a fetch protocol that avoids retrieving cover messages from idle channels. Crucially, both protocols are oblivious and do not reveal sensitive information to the service provider (e.g., which channels were active and carried real messages from buddies). Finally, Groove minimizes the cost that each message channel induces on the mixnet, with the most crucial improvement over prior work being the memory footprint. Groove supports 75M parallel message exchanges with 100 servers (prior work supports 1M–10M parallel message exchanges in the same deployment [16, 19, 20, 26]).

We analyze Groove's design and show that it achieves differential privacy against an attacker who has complete control over the network and has the power to compromise many servers that make up the system (including all service providers). Through this analysis, we choose the system's parameters that would provide a strong degree of privacy.

To demonstrate that Groove can support a large user base, we built a prototype and evaluated its performance on AWS servers distributed across four data centers. We also implemented a mobile client, deployed it on a Pixel 4 phone, and measured Groove in terms of battery and network usage. We show that Groove scales well with the number of servers and using 100 servers, it supports 1–3 million users sending and receiving messages from 50 buddies every 32–80 seconds. The client uses 54 MB–106 MB of network bandwidth per month and its battery consumption increases by 16% compared to the idle phone (when using cellular data). We recognize that Groove's latency is high compared to traditional messaging apps; nonetheless, it removes the rigid client requirements of previous designs and enables large-scale metadata-private messaging for mobile users.

In summary, our contributions are the following:

- oblivious delegation, an approach for offloading rigid mixnet requirements to an untrusted service provider, which enables client flexibility;
- Groove, a scalable metadata-private messaging system, and an analysis proving Groove's privacy guarantee;
- an implementation of Groove and an experimental evaluation of its performance, showing that it can support mobile clients and scale well to handle a large user base.

2 Related Work

Differential privacy. Groove's approach of providing differentially-private communication using a mixnet is based on prior work [19, 26, 27]. However, these earlier designs impose rigid requirements on users. First, users must be simultaneously online, and their clients must send and receive messages at every round to communicate. Second, users can only run a client on a single device, which also sidesteps the problem of device partitions. When a user's devices cannot coordinate, a conversation (which normally has one message per round) might end up with multiple messages per round, revealing the recipient. Groove addresses these problems through oblivious delegation.

Private Information Retrieval (PIR). PIR protocols support storing and retrieving information asynchronously from a database, without revealing anything about which message is retrieved [7, 23]. These protocols could provide a good degree of flexibility to clients: they have low client bandwidth requirements and allow the recipient to read a message at any time after the sender deposits it in a database. However, as users' messages accumulate in the database, processing each client's PIR query requires more work from the database server, limiting scalability. For example, Pung [2, 4] supports up to 20k–60k fetches per minute with 1M messages in its database. Express [11] optimizes PIR retrievals at the cost of making writes more expensive, but still the system only supports sending tens of messages per second across all clients. In contrast, Groove clients induce the same amount of work independent of how many messages accumulate in the system, enabling millions of users to send and receive messages every minute.

Communicating with many buddies. Metadata-private messaging systems minimize the load on the servers and overhead for the client by limiting users to receive messages from one buddy at a time [2, 4, 16, 19, 26, 27]. They require buddies to run a hefty dialing protocol to agree on when to communicate. For PIR systems, such as Pung and Addra [1,4], relying on dialing is even more essential; since reading is expensive for the servers, clients cannot retrieve messages from each buddy all the time. Many systems [1, 2, 16, 19, 26] propose using Alpenhorn's differentially private dialing [21], which takes about 5 minutes to coordinate between buddies and requires 62 GB of client bandwidth per month [21, §8.2]. Groove mitigates the clients' costs through its oblivious message fetching protocol and minimizes the load each message channel induces on the system, so users can keep channels with many buddies (e.g., 50 buddies per user in our implementation). Dialing may also be seen as a limited form of asynchronous messaging between users. The Vuvuzela [27] and Alpenhorn [21] dialing protocols use at least 3000× more bandwidth on clients than Groove, when configured to provide 1-minute messaging latency like Groove.

Metadata-private communication over persistent channels. Groove's users communicate over persistent message channels, similar to Tor's circuits. However, Tor does not protect users from a strong adversary model. An adversary monitoring the network can infer the path of users' messages across relays by correlating incoming and outgoing packets. Hydra [25] uses circuits to connect two endpoints, like Tor's hidden services [5], but does not hide the number of active conversations to the adversary [25, §4.6¶1], which allows for intersection attacks. Yodel [20] uses a mixnet with persistent circuits but requires two communicating users to be online at the same time to coordinate their circuits. Both Hydra and Yodel support only one device per user and have clients send and receive messages every round, two rigid shortcomings that Groove addresses.

Flexible clients through trusted servers. Loopix [24] hides metadata by relaying messages through a mixnet. It addresses the client flexibility problem by storing messages for users at *trusted* service providers. The service providers sees when a user receives a message, and can thus perform intersection attacks with other users who were online in time to send this message. Thus, users must make a tricky decision about which service providers they trust. Furthermore, the service provider is a singular target for learning about the relationships of its users. Pond [17] requires users to run a trusted server themselves. This leads to a simpler design than Groove (e.g., partitioned devices are not a concern) but limits the system to savvy users who can securely operate their own servers.[1] This is problematic for metadata-private communication, which benefits from a large user base [9]. In Groove, users maintain privacy even if their service providers fall under the attacker's control, so they do not need to operate servers themselves.

3 Overview

Groove allows buddies that share a secret to secretly exchange short text messages (e.g., 100B). Each buddy can run clients on multiple devices and seamlessly switch between them. Service providers bridge between the rigid mixnet protocol and the flexible clients, that can send and receive messages asynchronously. Each user designates a service provider and runs Groove's oblivious protocols for setting up message channels (Figure 1a), replacing messages (Figure 1b), and fetching messages (Figure 1c). The service provider submits messages to the mixnet on the user's behalf and stores messages that the user receives from the mixnet. It also helps the user synchronize their clients across all devices. Users

[1] Running a personal server in the cloud undermines security against a strong adversary that may be able to compel the cloud operator to disclose data or exploit side channels by deploying a VM on the same machine, and operating a physical server requires significant effort and expertise.

need not trust their service provider for privacy. To anonymize messages, Groove uses a parallel mixnet [12]. Parallel mixnets scale well with the number of servers by offering many parallel routes for processing messages. Groove extends the servers' message processing logic to enable oblivious delegation, and it is otherwise agnostic to the mixnet's internal design choices, such as the server topology or verification protocol (that ensures servers process messages correctly).

3.1 Threat model

Groove aims to hide its users' communication metadata from a *global active attacker* that controls all network links. In particular, this attacker observes when clients (dis)connect from the network and may partition users' devices. The attacker may also control all service providers, and each mixnet server with some probability f, which is provided as an assumption in the system's configuration. A smaller f yields better performance at the expense of a stronger trust assumption. The attacker can also run arbitrarily many clients. Still, an attacker could learn about a user's communication by directly compromising their device or the devices of their buddies [3], although forward secrecy prevents them from learning about the user's communication in the past. Finally, Groove assumes that the attacker is computationally bounded, so standard cryptographic primitives, such as hash functions and encryption schemes, are secure.

3.2 Goals

Privacy. Groove achieves differential privacy [10]. Consider an attacker and their view of the system through all network links and service providers, and the mixnet servers and clients that they operate. Groove ensures that the attacker's view is likely to be the same whether two users, call them Alice and Bob, are buddies or not. More formally, consider the attacker's observations $\mathcal{O}$ and the following two scenarios. In one scenario, Alice and Bob are buddies and can chat (denoted by $\mathrm{A} \leftrightarrow \mathrm{B}$), and in the other, they are not buddies and cannot chat (denoted by $\mathrm{A} \not\leftrightarrow \mathrm{B}$). Groove ensures the following inequalities for small $\varepsilon, \delta \geq 0$:

$$Pr[\mathcal{O}|\mathrm{A} \leftrightarrow \mathrm{B}] \leq e^{\varepsilon} Pr[\mathcal{O}|\mathrm{A} \not\leftrightarrow \mathrm{B}] + \delta, \quad (1)$$
$$Pr[\mathcal{O}|\mathrm{A} \not\leftrightarrow \mathrm{B}] \leq e^{\varepsilon} Pr[\mathcal{O}|\mathrm{A} \leftrightarrow \mathrm{B}] + \delta$$

That is, the probability for any observations the attacker could make is close under both scenarios (up to small constants ε, δ). Informally, Alice being buddies with Bob appears to the adversary almost as likely as them not being buddies. Thus, Alice could plausibly deny being buddies with Bob or claim to be buddies with anyone else. This privacy guarantee holds even if Alice or Bob is the sole honest user of a rogue service provider, the attacker partitions their devices, and observes the system for a long time.

Client flexibility. Groove should not impose strong timing or resource requirements on clients. It should allow Bob to retrieve Alice's message at any time after it reaches Bob's service provider and accommodate clients with network and battery constraints (that need to minimize communication or that might go offline). In particular, Groove should support clients running on mobile devices. At the same time, Groove should allow other clients to connect more often and achieve lower message latency. Finally, Groove should enable users to run clients on multiple devices and switch between them.

Performance. Groove aims to support millions of users with dozens of buddies. Once Alice sends a message, Bob can retrieve it with a latency on the order of a minute. The system should scale, i.e., provide the same performance to a larger user base by deploying proportionally more servers.

Availability. Groove's availability guarantees in the face of failing servers should primarily come from the mixnet and not degrade by its oblivious delegation approach. An overloaded or downed service provider can prevent service to its users, but the system's availability for users of other service providers should not be affected. Since the service provider is untrusted, it can use replication for better availability guarantees without security implications for its users. Moreover, having untrusted service providers also allows users to change to another provider if they suspect their provider is preventing service without risking exposing sensitive information to more parties (the current and new provider). We describe how users can detect and combat such providers in Groove's design (§5).

4 Background

Groove's users send and receive messages over message channels called *circuits*. A circuit is a fixed route of servers in the mixnet that persists for an *epoch*, which consists of many communication rounds (e.g., an hour to a day's worth). Each circuit connects a user's service provider to a *dead drop*, an ephemeral address where users exchange messages. Two buddies coordinate pseudorandom dead drops using a shared secret. The mixnet ensures that an adversary cannot correlate which service provider connects to what dead drop using noise messages. Groove borrows this communication model from previous systems [5, 19, 20, 25] (see §2), but changes the way circuits establish to support oblivious delegation. We summarize below the existing techniques that Groove uses to simplify the exposition of its new mechanisms in §5.

Like other mixnets [15, 26, 27], Groove's mixnet servers have unique public keys per epoch. Clients know the servers' public keys, e.g., through a transparent public key infrastructure [18, 22]. The way rounds are kicked-off depends on the mixnet, which Groove abstracts; e.g., many designs use an untrusted coordinator that announces to all mix servers when to start new rounds [15, 16, 19, 20, 26, 27].

Messaging over circuits and dead drops. The circuit setup message is onion encrypted with the public keys of the servers on the route. Each onion layer includes the next

server's ID, an ephemeral Diffie-Hellman public key, and an authentication code that ensures no earlier server has modified the onion. Each server completes the Diffie-Hellman handshake to derive a shared symmetric key with the client, and uses this key to verify the authentication code. After receiving messages from all servers in the previous hop, the server deduplicates and shuffles messages, and forwards them to the next hop. Servers store their shuffle's permutation and the symmetric keys; later, they use these records to process messages over the circuit.

During a communication round, mixnet servers process one message on each circuit. Each message is onion encrypted with the symmetric keys registered at circuit setup. The mixnet's servers shuffle and decrypt messages they receive using the permutation and symmetric keys they stored earlier. Servers deduplicate messages on the same circuit. If a server does not receive a message on a circuit, it fills in a cover message to ensure all circuits have one message. The symmetric encryption ensures that any random message that a server fills in is indistinguishable from a real message to other servers along the route. When two circuits connect to the same dead drop, the server hosting its address swaps the messages on these circuits and sends them back through the circuits, which is how buddies exchange messages.

Noise. Previous work shows how mixnet servers can add noise messages to protect metadata [19, 27]. We apply this technique in the context of Groove's circuits. For each inter-server mixnet link, each server decides on the number of noise circuits to route over that link by drawing from the Poisson distribution. Servers generate two kinds of noise: "doubles" which is a pair of circuits bound to the same dead drop to obscure the number of buddy-relationships, and "singles" which is a circuit terminating at a dead drop without a pair, to obscure the case where one of the buddies does not create their circuit to the shared dead drop. Like Karaoke [19], Groove uses Bloom filters to ensure that malicious servers do not drop the noise circuits.

5 Design

Figure 1 illustrates the parties in Groove: users, clients running on different types of devices, service providers, and the mixnet. When Alice and Bob become buddies, they add each other to their address books, and their clients establish a fresh shared secret. This secret allows Alice and Bob's clients to authenticate and encrypt messages (end-to-end) and coordinate dead drops for exchanging messages. The clients might create this secret out-of-band (e.g., by scanning QR codes if users meet in person) or via a metadata-private "add-friend" protocol (like in Alpenhorn's protocol suite [21][2]).

Users designate a service provider that stores their messages and participates in the rigid mixnet protocol on their behalf. The mixnet operates in rounds, where messages are exchanged over circuits. We envision rounds being relatively frequent, e.g., starting every 30 seconds to a minute. Every round, service providers submit messages to the mixnet. The mix servers shuffle messages and ensure each circuit carries precisely one message per round (by deduplicating messages on the same circuit and filling in a cover message when one is missing, as §4 describes). Messages are exchanged between circuits at the end of the mixnet and then sent back through the mixnet towards the service providers, where users can fetch their messages.

[2]Alpenhorn's add-friend protocol differs from its *dialing* protocol, which precedes every conversation—a cost that Groove avoids.

```
while true {
   // Block, waiting for the next wakeup event.
   <-client.Schedule // §5.1

   if oncePerDay {
      client.RefreshCircuits() // §5.2, §5.3 and Fig. 4
      client.ForwardSecrecy()  // §5.5
   }

   buddy, msg := client.OutgoingMessageQueue.Pop()
   // If the user has nothing to say, send cover traffic.
   if buddy == nil {
      msg = random.Bytes(MessageSize)
   }
   client.SendMessage(buddy, msg) // §5.4 and Fig. 5
   client.CheckForMessages()      // §5.4 and Fig. 6
}
```

Figure 2: The client's main loop. The client refreshes circuits once per day, at that time it also evolves the multidevice and buddy keys for forward secrecy. It sends and receives messages according to the schedule, which can be configured to balance battery life with communication rate.

In the remainder of this section, we introduce the concept of client schedules, present the protocols for oblivious delegation from Figure 1, and the mechanism for forward secrecy. Our descriptions follow the client's operation, outlined in Figure 2, and its interactions with the service provider via the API depicted in Figure 3.

5.1 Client schedules

To achieve Groove's privacy goal, the network traffic pattern between a user's client and their service provider must not reveal information about the user's communication with their buddies. In particular, this pattern includes when the client initiates requests to the service provider, which we call the client's *schedule*. The adversary can potentially infer any information that goes into deciding the client's schedule, and hence it should be independent of the user's buddy-relationships, which Groove aims to hide. Groove gives flexibility for clients to operate on their own schedule, independent from other clients. In this manner, it can accommodate clients on low-power devices with lightweight schedules without impacting other clients. A simple and safe schedule is to communicate with the service provider at regular intervals. Clients can use different intervals to trade network and power consumption for communication latency. They can also piggyback on other device wake-

```
var B = MaxBuddies
type Onion = [MessageSize + SymmetricOnionOverhead]byte

rpc BeginTransaction() *Txn
rpc (t *Txn) Commit() error

rpc (t *Txn) GetAddressBook() ([]byte, int)
rpc (t *Txn) SetAddressBook(data []byte, round int)

// RPC to set the circuit setup onions for an epoch.
// Each buddy corresponds to 2 circuits.
rpc (t *Txn) SetEpochOutgoing(epoch int, onions [B][2][]byte)

// RPC to fetch messages from our buddies.
rpc (t *Txn) GetHeaders(epoch int, round int) [B][2]byte
rpc (t *Txn) ShuffleInbox(ShuffleParams) [][]byte
rpc (t *Txn) FetchInbox(DHkey []byte, indices []int) [][]byte

// RPCs to queue a message for a buddy.
rpc (t *Txn) GetRoundOutgoing(epoch int, round int) RoundData
rpc (t *Txn) SetRoundOutgoing(epoch int,round int,rd RoundData)

type RoundData struct {
   // Messages are split into two, one for each buddy circuit.
   Onion [2]Onion
   // Previously sent message for this round.
   OutboundMsg []byte
}
```

Figure 3: Service provider API. Clients use these RPCs asynchronously to setup circuits to their buddies and send/receive messages through them.

ups (like checking for software updates) to interact with the service provider at a relatively low cost.

Clients can change their communication patterns if this change is independent of the user's buddies. For instance, it is safe for Alice's client to skip transmissions due to a network outage or because Alice boards a flight and her phone disconnects from the internet. It is also safe for users to have correlated schedules, as long as the correlation is not caused by their relationship status (being buddies or not). For example, users in the same time zone may prefer their devices to be more conservative during the day when on battery, but less at night when near a power outlet. Such correlations do not leak new information to the attacker (who can already observe their IP addresses and deduce their geographic locations). However, changes in the client's network patterns that depend on a user's buddies are unsafe; e.g., if Bob's client stops sending messages whenever Alice's device goes offline, an adversary might infer that Bob is connected with Alice.

5.2 Non-interactive circuit setup

Groove splits time into epochs, which correspond to the circuits' lifetimes. Periodically, e.g., once a day, clients call `RefreshCircuits` to generate circuit setup messages and upload them to their service provider (see the client's main loop in Figure 2). The service provider queues these messages and sends them to the mixnet at the appropriate time (sending circuit setup messages for the next epoch when the preceding epoch nears its end), even if all of the user's clients go offline. Users exchange messages over circuits,

```
func (c *Client) RefreshCircuits() error {
  epochs := c.serviceProvider.UpcomingEpochs()
  txn := c.serviceProvider.BeginTransaction()

  // Get address book and epoch of last update.
  addressBook, epochUpdated := txn.GetAddressBook()
  addressBookKey := c.MultiDeviceKey[epochUpdated]

  // Merge address books (buddy lists) across devices.
  prevBuddies := Decrypt(addressBookKey, addressBook)
  c.buddies = MergeAddressBooks(prevBuddies, c.buddies)

  // Pad the buddy list so its size doesn't reveal anything
  // to the provider and so we generate noise onions below.
  if len(c.buddies) < MaxBuddies {
      c.buddies = append(c.buddies, GenerateFakeBuddies())
  }
  newBook:= Encrypt(c.MultiDeviceKey[c.currentEpoch],c.buddies)
  txn.SetAddressBook(newBook, c.currentEpoch)

  for epoch := range epochs {
   var onions [MaxBuddies][2][]byte
     for i, buddy := range c.buddies {
        // Devices use the same PRNG to choose circuit
        // routes & tags, enabling deduping setup messages.
        randRouteTag := PRNG(i, epoch, c.MultiDeviceKey[epoch])
        onions[i][0] = GenCircuitSetupMsg(randRouteTag,buddy,0)
        onions[i][1] = GenCircuitSetupMsg(randRouteTag,buddy,1)
     }
     txn.SetEpochOutgoing(epoch, onions)
  }
  return txn.Commit()
}
```

Figure 4: Pseudocode for updating a client's address book and corresponding circuit setup onions for upcoming epochs, which are stored on the service provider. It is safe for multiple devices to run this function concurrently.

so adding or removing buddies only takes effect on epoch boundaries, when circuits are established. The more epochs a client prepares for in advance (by uploading circuit setup messages for future epochs in `RefreshCircuits`), the longer the user can go offline and keep receiving messages from their buddies. If all of a user's clients remain offline beyond this number of epochs, Groove will eventually not be able to establish circuits with the user's buddies, preventing them from communicating. Differential privacy is still maintained, however, due to mixnet noise during circuit setup. The epoch's duration is a knob that allows Groove to trade less client communication for higher latency in setting up circuits with new buddies.

Figure 4 gives the pseudocode for `RefreshCircuits`. First, the client synchronizes the user's contacts through the service provider, since the user might have added or removed a buddy through another device. The service provider's `BeginTransaction` and `Commit` APIs allow each of the user's clients to retrieve and upload data atomically with respect to the user's other clients, which may simultaneously call `RefreshCircuits`. Rogue providers can break the transactional semantics or deliver different address books to different clients, leading clients to set up circuits for stale address books; Groove protects against such providers, as we prove in §6. The client retrieves the address book from the

service provider, appends new buddies to the first available slots, and pushes the new address book to the service provider. Clients pad the address book to `MaxBuddies` slots and encrypt it under the multidevice key, which hides when users add or remove buddies.

Next, the client prepares two circuit setup messages for each buddy and many upcoming epochs (e.g., for the next month) and uploads these messages to the service provider. The reason for creating two circuits per buddy, rather than one circuit, is to allow clients to fake connections to buddies when the user has less than `MaxBuddies` friends, hiding their number of friends. In this case, the client creates two "cover circuits" to one dead drop, so there are precisely two circuit setup messages to all dead drops a client uses (regardless of the number of buddies the user has). The client learns the epoch number from the service provider and relies on it to submit circuit setup messages at the right epoch; however, the provider can cheat. Thus, the client writes the epoch number in each onion layer, so the mixnet servers can discard circuit setup messages the provider sends at the wrong time. As one epoch nears the end, the mixnet runs circuit setup for the next epoch, so circuits are ready at all times to route messages between buddies. The mixnet's servers then process the setup messages as in previous works (§4).

One issue is that Groove must ensure privacy when one user establishes a circuit and their buddy does not. Groove handles this challenge by noising the circuit setup step with cover circuits generated by the mix servers. The cover circuits ensure that, regardless of whether Alice and Bob are buddies, the attacker observes the same traffic pattern (on the network and to dead drops). Groove applies Karaoke's noising technique [19], of creating "single" and "double" dead drop accesses, to circuit setup messages (summarized in §4).

Circuit setup messages in Groove are acknowledged to the clients, which then learn whether anyone dropped their circuit or the circuit from their buddy. This allows users to detect active attacks and, as in previous systems [19,20], provision a tighter privacy budget when choosing the system parameters (and thus, achieving better performance) compared to systems that do not detect active attacks (like [26, 27]). To achieve this in Groove, the content of circuit setup messages is a pseudorandom ID that each user derives from the secret they share with their buddy (or the multidevice key if the circuit is cover). When the circuit setup message reaches the dead drop, the server hosting that dead drop swaps the content of messages (see §4) and returns it through the mixnet to the user's service provider. The next time the client connects to the provider and learns the current epoch, it downloads the returned IDs of the circuits from previous epochs and checks they are correct by deriving the IDs the buddies would use. Correct IDs acknowledge to clients that the circuit setup message from them and their buddy had propagated to the dead drop, so all servers in the route shuffled these messages with the other setup messages.

5.3 Oblivious replacement

Preparing circuits for future epochs allows users to go offline for a long time. However, users might add buddies after submitting circuit setup messages. Groove clients can update the circuit setup messages stored at the service provider, so users can communicate with new buddies soon after adding them to their address books. Each client performs this replacement periodically, according to its schedule (Figure 2); if there are no changes in the address book, the client uploads fresh circuit setup messages pointing to the same dead drops. The key challenge in performing this replacement is that, without coordination across the user's devices, several of their clients might establish circuits to the same dead drop. This will create a distinct access pattern (i.e., not the single- or double- dead drop access patterns covered by the noise). An attacker controlling the dead drop's hosting server can then associate that dead drop with the user.

Groove introduces circuit tagging, which enables safe replacement of old setup messages without relying on communication between a user's devices, as illustrated in Figure 1b. When choosing the circuit's path, `RefreshCircuits` seeds a pseudorandom number generator (`randRouteTag` in Figure 4) for each epoch with the multidevice key and the buddy's slot number in the user's address book. This pseudorandom number generator is the same across all of the user's devices. The clients then use it to choose the route for each address book slot and include a pseudorandom tag derived from this generator in each layer of the circuit's setup message. Honest servers on the route deduplicate circuit setup messages according to this tag. Although the routes are the same, each client submits different-looking messages to the service provider since the onion encryption scheme is randomized.

This route and tag selection procedure ensures that Alice submits circuit setup messages with the same route and tags across all her devices for each buddy in her address book. If all of Alice's devices upload circuit setup messages and a malicious service provider submits all of them, the first honest server along each the (identical) routes of duplicate messages observes the duplicate tags and discards the redundant messages. This ensures the user's circuits do not access a dead drop an unusual number of times.

5.4 Efficient messaging

Groove uses oblivious protocols for efficiently submitting and fetching messages over many concurrent circuits.

Sending messages. During an epoch, there are $2 \times$ `MaxBuddies` circuits available to the client for messaging. Uploading a message to every circuit every time the client's schedule is triggered (Figure 2) incurs unnecessary bandwidth costs, especially since users do not typically talk to all of their buddies at once. Instead, the Groove client submits just one message (split into two parts, leveraging two circuits

```
func (c *Client) SendMessage(buddy, msg string) error {
   epoch := c.currentEpoch
   round := c.nextRound

   txn := c.serviceProvider.BeginTransaction()
   rd := txn.GetRoundOutgoing(epoch, round)

   prevMsg, prevBuddy :=
      Decrypt(c.MultiDeviceKey[epoch], rd.OutMsg)
   if IsRealMessage(prevMsg) {
      err = "refusing to overwrite user-typed message"
      // Re-encrypts previous content.
      msg, buddy := prevMsg, prevBuddy
   } else {
      msg = MsgHeader(epoch, round, buddy.key[epoch]) ++ msg
   }

   msg1, msg2 := SplitMessage(msg)
   ix := GetBuddyIndex(buddy)
   onion1:= EncryptSymOnion(epoch.circuits[ix][0].keys,msg1)
   onion2:= EncryptSymOnion(epoch.circuits[ix][1].keys,msg2)

   txn.SetRoundOutgoing(epoch, round, RoundData{
      Onion:  {onion1, onion2},
      OutMsg: Encrypt(c.MultiDeviceKey[epoch], {buddy, msg}),
   })

   return txn.Commit(), err
}
```

Figure 5: Client pseudocode for sending messages.

per buddy) and does not reveal the designated circuits to the service provider. The service provider then broadcasts the message on all of the user's circuits (i.e., the first half of the message on even circuits, and the second half on odd circuits). Messages are encrypted end-to-end, so only the intended recipient can decrypt them. If a rogue provider does not broadcast a message, the first honest mix server on each path will fill in a cover message, ensuring buddies keep receiving messages at the same rate (§4). Figure 5 gives the client's pseudocode for sending messages.

Fetching messages. The recipient's service provider receives messages from the mixnet (one message per circuit per round) and stores them for the clients to fetch later. Clients should avoid fetching cover messages to minimize their bandwidth and energy costs (e.g., messages that mixnet servers fill in or that are intended for another buddy), and at the same time, hide which circuits carry real messages to hide when someone messages the user. We could use PIR to fetch messages, but this comes at high cost and complexity clients and the service provider, especially as messages accumulate. Instead, Groove's fetch protocol relies on mixing messages, where a set of servers processes the messages from the provider just once regardless of the number of messages a client retrieves. We describe it following the illustration in Figure 1c and the pseudocode in Figure 6.

When Bob's client calls `CheckForMessages` from its main loop (Figure 2), it first retrieves from the service provider a short header for each stored message (e.g., two bytes per buddy). The header acts as a pseudorandom flag, shared between the two buddies: when Alice sends a message to

```
func (c *Client) CheckForMessages(epoch int) ([]int,[][]byte){
   round := c.GetNextRound()
   mixers := RandomMixnetPath(11) // Path of length 11

   hdrs := c.serviceProvider.GetHeaders(epoch, round)

   // Identify the indices of real messages from buddies
   indxs = []
   for i, buddy := range c.buddies {
      if hdrs[i] == MsgHeader(epoch, round, buddy.key[epoch]){
         indxs = append(indxs, i)
      }
   }

   // Shuffle user's inbox with a fresh key
   pk, sk := GenerateDHKeypair()
   nonces := c.serviceProvider.ShuffleInbox(ShuffleParams{
      Epoch:          epoch,
      Round:          round,
      PublicKey:      pk,
      Mixers:         mixers,
   })

   // Predict the indices after mixing step in ShuffleInbox
   shuffledIxs := PredictPositions(sk, mixers, nonces, indxs)

   // Ensure we fetch a constant number of messages
   PadWithFakeRequests(shuffledIxs)

   // Fetch messages at the (shuffled) indices
   onions := c.serviceProvider.FetchInbox(shuffledIxs)

   // Remove onion encryption from ShuffleInbox's mixing step
   msgs := DecryptOnions(onions, sk, mixers, nonces)

   // Map the messages back to the correct buddy
   Unshuffle(msgs, sk, mixers, nonces)

   return indxs, msgs
}
```

Figure 6: Client pseudocode for oblivious fetch. Clients first fetch headers from the service provider and identify indices of interest. Finally, they request the shuffled indexes corresponding to the result of the mixing step of the oblivious fetch.

Bob, her client derives the header's value from the current round and the key that Alice and Bob share, and sends it along with the content (inside the onion). Bob's client derives the same header and compares it against the header from Alice's circuit (users only need to check one of the circuits in the pair). If the header values match, his client knows to fetch the corresponding message next. To avoid revealing messaging rates between buddies, the service provider must not learn from which circuits Bob fetches messages. Groove hides this by mixing Bob's messages again, as follows.

`CheckForMessages` instructs the service provider to submit all messages pending for Bob to a "fetch mixchain," which is a sequence of mixnet servers chosen by the client. The client also supplies a new Diffie-Hellman public key, which the service provider relays to the first server in the sequence. The first server uses its secret key to complete the Diffie-Hellman handshake and derives a shared secret with the client; it then chooses a fresh nonce and hashes it with this shared secret to derive an ephemeral symmetric key. The server derives the shuffle permutation from this ephemeral

key and encrypts the messages. The server passes its nonce, the client's public key, and the list of the remaining servers to the next mixchain server, which continues in the same fashion. The nonces ensure that a mixchain server's output looks freshly random, even if a rogue service provider replays an old request. The last server passes the shuffled messages and the nonces to the recipient's service provider, who in turn forwards the nonces to the client. The client then derives the symmetric key for each server and computes the (shuffled) position of the messages it should fetch. Finally, it fetches messages at the shuffled indices directly from the service provider. The freshness of the client's Diffie-Hellman key ensures that it accesses random-looking locations each time it runs the protocol.

Clients download a small fixed number of messages in every round. For example, a client could always download six messages per round to support up to three buddies simultaneously messaging the user. This way, the number of messages clients retrieve does not reveal information about messaging rates to the attacker. The oblivious fetch mechanism also lets clients retrieve a different number of messages to quickly catch up after being offline for an extended time. Similarly, if there's a burst of real messages in some round (beyond the fixed fetch rate), the client can run a large daily fetch procedure (e.g., for 100 messages) to catch any missed messages at relatively low cost.

5.5 Forward secrecy

Groove achieves forward secrecy for both the message contents as well as metadata, meaning that an adversary that compromises a user's device cannot retroactively decrypt messages, determine who the user communicated with, or who was in the user's address book. Note, however, that an adversary that compromises a user's device does get to see the device's current address book and messages.

The challenge in achieving forward secrecy in Groove is that user devices may be partitioned from one another, and thus cannot refresh their keys by coordinating over the network. To achieve forward secrecy in this setting, Groove deterministically evolves secret keys based on the epoch: for each passing epoch, clients hash the keys (as described below) and erase the pre-image. Clients evolve the multidevice key, ensuring that an adversary compromising a device cannot track old circuit routes from the user to a dead drop, and cannot decrypt old copies of the user's address book. Clients also evolve shared secrets across buddies to ensure that the adversary cannot decrypt old messages.

Deterministically evolving keys could allow an adversary that obtains old keys (e.g., by compromising a user's old powered-off phone) to derive all future keys, thereby compromising data and metadata privacy for all epochs since the stale compromised key, a form of post-compromise insecurity [8]. To avoid this vulnerability, Groove involves the servers in computing the hash function for evolving keys, and honest servers refuse to compute this hash function for epochs in the past or that are more than T epochs in the future. This limits the vulnerability window by ensuring that an adversary with access to older keys cannot evolve them forward to decrypt newer user data or metadata.

Realizing this approach is challenging because it requires combining secrets from two parties, without either party learning the other's secret. The client wants to hash a key without giving it to the server, and the server must not reveal its secret that would allow hashing arbitrary values in the future. We address this by using an oblivious pseudorandom function (OPRF), specifically, the verifiable DH-OPRF construction from [13]. Groove's clients run the verifiable DH-OPRF protocol with each server (evolving their keys with each server in turn). This ensures that the keys are evolved with at least one honest server's secret key. Clients verify the DH-OPRF result against the server's public key for that epoch (§4). Verifying this result is critical here, since it ensures that an adversary cannot cause two of a user's devices to diverge in their multidevice key, which would cause them to create different circuit paths and thus leak metadata.

The client evolves keys every day (see Figure 2), deleting keys older than T epochs from the newest key. The client keeps keys for T epochs in the future, which allows circuit setup messages to be prepared in advance. If the device is off for longer than T, the client's newest key becomes stale, and the servers will refuse to roll it forward with their old keys. Thus, the duration T is a tradeoff between security and user convenience: after T epochs offline, a device must be set up again manually (e.g., by copying keys from another device). If the provider lies about committing the address book, then the evolved key on the device is also useless since it cannot decrypt the address book.

5.6 Provider availability and switching providers

Groove's service providers are *not* trusted for privacy (as we prove in §6), but a service provider can still block communication for its users. To address this, Groove clients can periodically send messages to themselves (on empty buddy circuits) and detect provider malfunction in case these messages often do not route back intact (the provider cannot tell which message is for a buddy and which would route back to the user). In this case, the client notifies the user to switch providers. Such self-addressed messages were proposed in prior work to detect attacks [24]. Keeping providers untrusted for privacy, however, simplifies dealing with such availability attacks compared to prior work since users can switch providers without risking exposing their communication metadata to more parties. Moreover, this property also protects against attacks that steer users towards corrupt providers and contrasted against systems with trusted providers (§2).

Users can also submit copies of messages to multiple providers to ensure availability when all but one provider

fail. This is safe since Groove ensures privacy even if rogue providers duplicate messages (by deduplicating messages in the mixnet, §5.3). One detail when using multiple providers is that Groove delivers only one message copy (to defend against malicious providers submitting multiple messages). Malicious providers can thus actively try to prevent service by submitting corrupt messages, hoping their copies will prevail. Users can detect this intervention and switch providers (as discussed above).

6 Privacy analysis

Clients send and receive messages through the user's service provider. They communicate according to a schedule that is independent of their buddy relationships (§5.1); thus, the clients' network pattern does not leak sensitive information about the user.

Since it places the users' trust in the mixnet's servers as a collective rather than their service provider, our analysis focuses on reducing the security of Groove to the security of the mixnet (§6.1). We analyze each oblivious protocol and show that a more restricted attacker, who controls the network and the same mixnet servers but not the service provider, can obtain the same information. Groove's design uses a parallel mixnet as a black box that hides the sender of messages. The dead drop-based message exchange provides differential privacy for every epoch (since users can change their communication patterns by setting up circuits on epoch boundaries), similar to prior work [19, 26, 27]. The advanced composition theorem [10, 3.20] allows to compute Groove's privacy guarantee after multiple epochs, as we do in §7.

6.1 Oblivious delegation

We now prove that the security of Groove reduces to the mixnet; controlling the service providers does not enable an attacker that already controls the network and some mixnet servers to learn new information. In particular, this implies that it is safe to be the sole user of a rogue service provider.

Theorem 1. Consider Groove's attacker, who controls the users' service providers, the network, and a portion of the mixnet servers, and his observations about the users' communication. A restricted attacker, who only controls the network and the same mixnet servers (but not the service providers), can obtain the same observations.

Proof. We consider all the ways a user's clients interact with their service provider. These interactions take part in three oblivious protocols (non-interactive circuit setup, oblivious replacement, and efficient messaging). We analyze each protocol in §6.1.1 – §6.1.3 and show that Groove's attacker cannot learn any information that the restricted attacker could not obtain (e.g., by dropping network packets). □

6.1.1 Non-interactive setup

The user's clients synchronize address books through one service provider. Clients always update the user's address book before preparing circuit setup messages (§5.2). On each update, the client uploads the address book under fresh encryption, padded to a fixed length (`MaxBuddies`), so the service provider cannot tell whether it has changed. The service provider may give stale address books to clients; this is our focus in §6.1.2.

Clients also retrieve the current epoch number from their service provider before preparing circuit setup messages. A malicious service provider can lie about the epoch number or submit the circuit setup messages to the mixnet at the wrong epoch. The client writes the epoch number in every onion layer of the setup message, and honest mix servers discard onions with the wrong epoch number. Thus, if an honest server exists en route, this service provider's attack is equivalent to a network attacker simply dropping the user's circuit setup messages. If there is no honest server on the route, then even the restricted attacker can learn everything about the circuit by observing the setup message going from one malicious mix server to the next. (Groove mitigates this risk by using sufficiently long mixnet routes, §7.1.)

6.1.2 Oblivious replacement & device partitions

A rogue service provider may interfere when clients replace circuit setup messages, or collect and submit setup messages from multiple devices. Since the attacker controls the network, it might partition devices and prevent them from communicating. The risk with partitioned devices is that a rogue service provider submits circuits from different devices and creates distinct dead drop access patterns (i.e., a dead drop getting more than two accesses, which is not obscured by the "single" and "double" noise). Groove solves this problem with its mechanism for choosing circuit routes and tagging circuit setup messages (§5.3), as we prove next.

Consider the user Alice with two partitioned devices, d and d', and a circuit they establish for the buddy at slot $i \in [1, \texttt{MaxBuddies}]$ in their respective address books for the same epoch. Both devices submit setup messages for circuits with the same route and tag: they derive them from the buddy's slot number i, the multidevice key, and the epoch number, which are all the same for both devices, even if their address book differs and have different buddies for the same slot (see Figure 4). (The multidevice key is identical on all devices for the same epoch. If two devices use different epochs, the mixnet will discard the circuit setup message from the device using the wrong epoch, as described above.) If there is no honest server on this route, then the attacker can trace Alice's messages through the malicious mixnet servers to the dead drop, regardless of controlling her provider.

Otherwise, an honest mix server exists on the circuits' route. It will de-duplicate the two circuit setup messages and ensure that only one circuit will be established. There

are two cases regarding the dead drops at the end of these circuits. First, the circuits from d, d' reach the same dead drop. In this case, since the honest server drops one message, the attacker's observations will be precisely the same as if the devices could coordinate and only one device submitted setup messages. The second case is that device d submits a circuit setup message to a different dead drop than device d'. A device-partitioning attacker can cause this, e.g., by giving one device an old address book where a now-occupied slot was free. In this case, one of the circuits' dead drops receives one less circuit, i.e., becomes a "single"-access dead drop, which is covered by Groove's noise. The attacker could obtain the same view by dropping one circuit setup message from Alice on the network (even if Alice had just one device).

6.1.3 Efficient messaging

The sender's client submits one fixed-length, onion-encrypted message to the service provider. However, it does not contain any information about the intended recipient and, therefore, cannot teach the provider about the user's communication (the service provider broadcasts it on all circuits, §5.4).

The service provider serves messages to the recipient's client by routing them through the fetch mixchain, which the client chooses when calling `CheckForMessages` (see §5.4). The choice of mixchain servers is independent of the user's buddy-relationships, so it leaks nothing about them. Since mixchain servers choose fresh nonces when shuffling messages, they uses different ephemeral keys for processing every fetch. Thus, the messages output from the fetch mixchain always appear random to the recipient's service provider. Furthermore, the client chooses a new ephemeral key each time it calls `CheckForMessages`, so it always fetches messages from random-looking locations (even if the service provider replays old headers). Therefore, having one honest server on the fetch mixchain ensures that the client's pattern of retrieving messages appears random every time.

7 Implementation

We implemented a prototype of Groove in Go on top of Yodel's mixnet framework [20] in 20k lines of code. The mixnet has a full-mesh server topology [19, 20], which allows Groove to scale with the number of servers. The prototype uses ChaCha20 for symmetric onion encryption, the NaCl box primitive to generate circuit setup onions, and the Blake2b hash function. To implement forward secrecy with DH-OPRF, we use BLS12-381 in the CIRCL library [6].

Communications between clients, service providers and mixnet servers all use gRPC over TLS 1.3 for transport security. The service provider uses BadgerDB to implement atomic transactions and manage user state.

Our implementation includes two types of clients, for desktop and mobile devices. The desktop client is a command-line program, and the mobile client is built for Android using the `gomobile` tool.

Memory usage. A prominent challenge in implementing Groove has been minimizing the circuits' memory footprint to allow users receive messages from any of their buddies in parallel. With 3M users, 100 circuits per user, 100 mix servers, and 14 mixnet hops, each server needs to keep track of at least 42 million pieces of cryptographic state per epoch:

$$100\,\text{circuits} \times 3\,000\,000\,\text{users} \times \frac{1}{100\,\text{servers}} \times 14\,\text{hops} = 42\text{M}.$$

Initially, we used AES-GCM to implement Groove's circuits, but its state is 512 B, resulting in at least 20 GB of memory usage per mixnet server. To reduce memory, we replaced AES-GCM with ChaCha20, which requires only 32 B per state, reducing this memory usage to 1.3 GB per server, but increasing CPU usage due to lack of hardware acceleration.

7.1 Parameter selection

We set Groove to resist $f = 20\%$ malicious mixnet servers, and the mixnet path length to be 14 hops (similar to prior work with the same mixnet topology [19, 20]). As shown in prior work, this topology requires two honest servers on a circuit's route, and the probability that this holds for all four circuits that two buddies use to communicate in Groove is $\geq 1 - 4 \cdot 10^{-8}$, assuming $f = 20\%$. The fetch mixchain requires only one honest server on its path, so we set its length to 11, which fails with even smaller probability. Users send 128-byte messages, split over the two circuits they create per buddy. Out of the 128 bytes, 2 bytes are reserved for the pseudorandom header indicating whether the message is real or cover traffic (§5.4), 12 bytes are reserved for the end-to-end authentication code, and 12 bytes are reserved for a nonce. Thus, recipients get a 102 byte encrypted text message per buddy per mixnet round.

Noise in practice. We configure Groove to tolerate 245 epochs of active attacks and 9600 epochs of passive observations. These parameters are comparable to the suggested configuration in Karaoke [19], which resists the same number of active attacks, and sustains passive attacks for a year (if epochs are at least 1 hour long, then 9600 epochs are over a year's worth).

Specifically, with 100 mixnet servers, each honest server creates 88 000 noise circuits on average in every epoch to provide $(\varepsilon = \ln 2, \delta = 10^{-4})$-differential privacy (the same ε, δ as Vuvuzela's implementation, and better than Karaoke's $\varepsilon = \ln 4, \delta = 10^{-4}$ and Stadium's $\varepsilon = \ln 10, \delta = 10^{-4}$ [19, 26, 27]). The more servers there are, the less noise each server contributes (e.g., a mixnet with 50 servers requires each of them to create 125k noise circuits for the same privacy level).

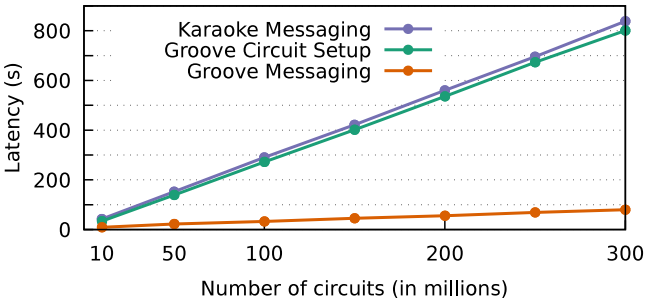


Figure 7: Latency of circuit setup and messaging rounds with respect to the number of circuits. The figure also compares with Karaoke's messaging round as a baseline (1 user relationship in Karaoke equals 2 Groove circuits) [19]. The mixnet has 100 servers.

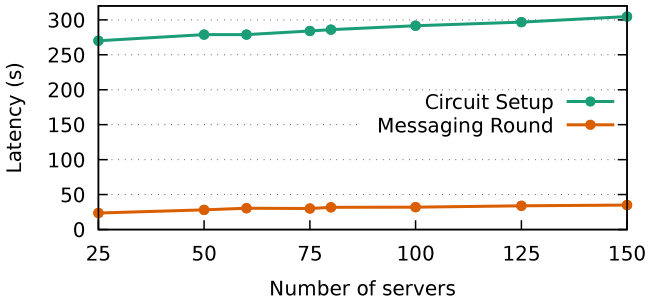


Figure 8: Scalability of the mix servers. The mixnet server load is constant (1 million circuits per server). This experiment shows that by having more mix servers, Groove can support more clients with the same communication latency.

8 Evaluation

We use the prototype to evaluate Groove's performance and costs. Our experiments answer the following questions:

1. What throughput and latency can Groove achieve?
2. How does Groove scale with the number of servers?
3. What are Groove's deployment costs?
4. What are the costs for a mobile client in terms of battery consumption and network usage?
5. What is the cost for a client catching up to old messages after having been offline?
6. How does Groove's performance compare to prior work?

First, we focus on the Groove's servers, i.e., the service providers and mixnet, and then on the mobile client.

8.1 Server performance and costs

Setup. We test Groove with 25–150 mixnet servers that we deploy evenly split across 3 EC2 regions across the US and one in Europe: `us-east-1`, `us-east-2`, `us-west-2`, `eu-west-1`. Each server is an `r5.8xlarge` VM with an Intel Platinum 8000 3.1 GHz CPU with 32 cores, 256 GB of memory, and a 10 Gbit/s network link. We evaluate with a single service provider on `us-east-1`: since service providers only buffer and relay messages, but do not participate in processing messages through the mixnet, the performance of Groove's mixnet does not depend on the number of service providers. Only clients connected through an overloaded service provider will experience performance issues.

We simulate hundreds of millions of circuits by having mix servers create extra circuits. Assuming each user has up to 50 buddies, and hence requires 100 circuits, the system load corresponds to relationships between millions of users. Although clients do not use these circuits, they correspond to real conversation load on the servers. We set Groove's system parameters as described in §7.

Throughput and latency. We measure messaging latency for a given circuit load in deployment of 100 mixnet servers. To measure the latency, we measure the time from when the service provider submits a message until the mixnet completes the round and returns the result to the service provider, plus the time needed for the fetch protocol (§5.4). Users will experience additional latency depending on their schedules and the network RTT to their service provider. Groove is not tied to one configuration of buddies per user, so we use the number of circuits to quantify Groove's load (each buddy requires two circuits). Since Groove ensures there is one message delivered on every circuit in every round §4, the number of circuits sets the load on its servers.

In Figure 7, we observe that the Groove's messaging round latency is 32.4 s, 55.9 s and 79.83 s for 100M, 200M and 300M circuits, respectively. The measured latencies have two components: The first, larger, component is mixing the users' messages and routing them from the source to the destination service provider; this represents the majority of the duration of Groove's messaging round (28.7 s, 50.8 s and 71.3 s for 100M, 200M and 300M circuits). The second, smaller, component ($\approx 11\%$ of the total duration) is the time spent on running Groove's fetch protocol (§5.4). For this second part, we model an average load of clients running the fetch protocol: we simulate fetches at the end of each round, and we wait for all messages of a round to be fetched before moving on to the next round. Thus, assuming the average user sets up 100 circuits (to communicate with up to 50 buddies), Groove could support 1M–3M users with these latencies.

The duration for setting up the circuits in Groove is 13.3 minutes for 300 million circuits (Figure 7). Since circuit setup is relatively infrequent (e.g., once a day), it can run in the background and stretch up to an epoch's duration. Groove's circuit setup is similar to a messaging round in Karaoke [19], which also offers differential privacy, though there is a difference in the payload size (Groove's payload is about 200 bytes smaller than Karaoke). For the same setup, where the system processes 300M messages per round, a communication round in Karaoke takes 14 minutes (Figure 7).

Scalability. We test how Groove scales with the number of users by measuring the latency for varying deployment sizes of 25–150 mixnet servers and keeping the number of circuits per server constant at 1M (so the load on the system increases proportionally to the number of servers). Figure 8 shows that

Groove scales well: it can support additional users at almost the same latency by proportionally increasing the number of servers. We attribute the slight latency increase to the fact that shuffling messages together requires each server, in every hop along the mixnet route, to wait for inputs from all other servers that processed messages at the previous hop.

Deployment costs. With 300 million circuits, each of the 100 mixnet servers sends at about 2 Gbps at peak usage, for a network usage of 13.4 GB per messaging round. In this deployment, setting up circuits for one epoch uses 47.5 GB of network data per mixnet server.

Service providers buffer messages for their users. For a user who generates circuit setup messages for the next 30 epochs (i.e., the next month if epochs last a day), the storage requirement for circuit setup is 2.1 MB. Further, if messaging rounds happen every minute, then storing a month's worth of received messages amounts to 264 MB per user.

Forward secrecy. On the server, computing OPRF incurs low overheads. A single server can answer 12k DH-OPRF requests per second, or 10^9 per day. A client evolves keys with all their buddies and the multidevice key every epoch (§5.5), creating 51 requests/day with day-long epochs. Servers can run the OPRF computations in the background and load-balance client requests throughout the day, allowing servers to easily support 1–3M users as in the earlier experiments. On the client-side, it takes 2.03 s to run `ForwardSecrecy` with 100 servers (primarily due to the network latency, then to the two pairing operations used in the verification), and the bandwidth usage is 150 kB/day.

8.2 Mobile clients

We evaluate Groove's mobile client, which represents an important class of clients enabled by its flexibility. We focus on two metrics: the impact on battery life and network usage. We run the mixnet with 32s messaging round time, which correspond to the latency with 100M circuits and 100 servers (Figure 7). We evaluate the mobile client on a Pixel 4 cellphone running the stock Android 10 OS. Our tests include cellular (3G with HSDPA) and WiFi networks.

Battery usage. We explore the impact of different schedules on battery life. To evaluate battery consumption, we connected the mobile device to a USB power meter (UM25C) and collected energy consumption roughly every 1 second.

We force-enable *doze mode* to reduce noise from apps running in the background; this is the battery-saving optimization that typically runs when the device's USB port is unplugged. Yet, running Groove's app in this mode implies that the client's transmissions may change without adhering to the schedule. Therefore, we excluded our app from doze mode (Android's `AlarmManager` provides APIs to avoid it).

First, we measure the baseline energy consumption when the phone is fully charged, idle, the screen is turned off, and Groove's client is not installed. We observe that after an hour, the idle phone consumes about 310 mWh of energy, both when the phone uses WiFi and cellular networks. Then, we run Groove's client with different schedules. At the beginning of the experiment, the client uploads circuit setup messages for the next 30 epochs. Then, the client sends a message and downloads pending messages according to its schedule (following Figure 2).

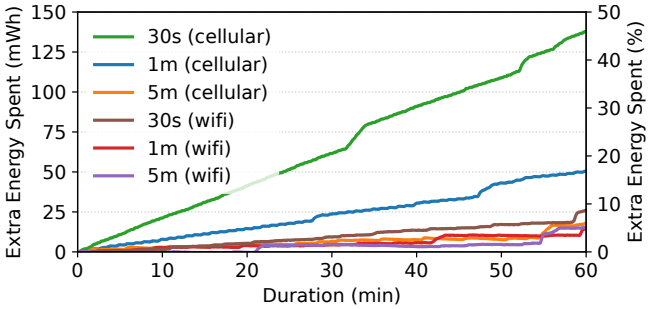


Figure 9: Client's energy consumption on a Pixel 4 phone for different schedules and network types. The graph shows how much more energy is spent running a schedule compared to leaving the phone idle. The bumps in some of the lines correspond to wake-ups to service other running apps (as we verified by reading device logs).

Figure 9 shows the energy consumption over an hour for different schedules compared to the baseline. It is apparent from the figure that as the schedule becomes more frequent, the energy consumption increases, especially when using cellular networks. A mobile client following the mixnet's $\approx$ 30s round-schedule would have increased battery usage by 47% over the idle phone, but flexibility allows the mobile client to use lighter schedules. On a 1-minute schedule, the client on the cellular network uses an extra 50 mWh compared to the baseline, a 16% increase. Moreover, the energy cost from running the client is substantially reduced further when the client runs on a 5-minute schedule (about 6% over idle). We hypothesize that this allows the phone to hibernate and save power. The energy consumption is more modest when using WiFi, which is more energy-efficient [14].

Network usage. We monitored the link between the client and service provider to quantify the client's network usage. It uploads one message to the service provider every time the schedule triggers and downloads a message per round (see Figure 2). Consequently, different schedules affect the message upload volume, which ranges from 33 kB/h with a 5-minute send schedule to 77 kB/h with a 1-minute schedule. The download volume is 39 kB/h on the 5-minute schedule and 69 kB/h on the 1-minute schedule. In addition, the client sends about 110 kB for the circuit setup per epoch; for a month's worth of prepared circuits, this phase costs 3.2 MB. These measurements show that Groove's client uses a total of 54 MB to 106 MB of bandwidth per month, depending on the schedule. We believe that Groove's moderate network usage is compatible with mobile data packages.

Catch-up. Groove's oblivious fetch protocol filters cover messages, allowing clients to quickly catch up on messages that buddies send their users after being offline for an extended time (§5.4). Consider a client that catches up on a month by downloading 500 messages per offline day (15k total); we evaluate it requires 11 MB of bandwidth.

8.3 Comparison with prior work

Groove provides asynchronous messaging with many buddies, whereas recent mixnet-based work, like Karaoke, Stadium, and Yodel [19, 20, 26], provide synchronous communication with one buddy. PIR-based approaches, like Pung [4], could allow asynchronous messaging but with a significant performance cost compared to [19, 20, 26] (owing to stronger privacy guarantees and weaker trust assumptions); see discussion in §2.

Thus, these prior systems rely on a hefty dialing protocol (Alpenhorn [21]) to synchronize between buddies before they can communicate. Groove outperforms prior designs when users talk with multiple buddies since dialing through Alpenhorn adds about 5 minutes of latency when users switch between buddies. On the other hand, if users communicate with just one buddy who is simultaneously online, they can avoid frequent dialing, and in this case, Yodel and Karaoke outperform Groove.

In more detail, Stadium, Karaoke, Yodel, Pung, and Groove were evaluated on a similar 100 server configuration, which we use to compare. We assume that users in Stadium, Karaoke, Yodel, and Pung only chat with one buddy that is simultaneously online. Karaoke supports 1M users with 7 s of latency (Figure 6 in the Karaoke paper [19]), while 1M Groove users can communicate with 50 buddies with a latency of 32 s. The increase in latency is only 4× that of Karaoke, despite Groove supporting all 50 buddies to message the user at the same time, since Groove establishes circuits through the mixnet (allowing for more efficient symmetric onion encryption rather than the public-key onion encryption used in Karaoke). Stadium induces latency on the order of minutes (see Figure 9 in the Stadium paper [26]) largely because of its use of zero-knowledge proofs to ensure that mixnet servers process messages correctly. Pung's latency increases quadratically with the number of users [2, 4]; with millions of users, latency is over 30 minutes (as we interpolate from Figure 8 in the Pung paper [2] assuming that a 100-server Pung cluster performs 100× better than one server). This performance gap grows with the size of the user base. Yodel builds on its interactive circuit establishment protocol to directly connect (not through the mixnet) each user to its buddy's dead drop. Groove avoids this direct connection to protect the user's communication metadata-privacy if the buddy is offline (hiding the user was trying to connect with an offline buddy). For 1M users, Yodel's latency is 750 ms (Figure 10 from the Yodel paper [20]), 42× better than 1M Groove users that can communicate with 50 buddies simultaneously (corresponding to the 100M circuits datapoint in Figure 8). If we limit Groove to allow one buddy per user, then it needs to support only 2M circuits per 1M users (as in Yodel), and Groove's latency shrinks to about 4× that of Yodel. This remaining performance gap is primarily due to Groove connecting both buddies to dead drops through the mixnet (above) and Groove's fetch protocol that conserves client bandwidth at the expense of routing the messages buffered at the recipient's service provider again through the mixnet.

Another significant difference is the client bandwidth. Dialing through Alpenhorn requires clients to receive 62 GB per month (§2), on top of the overlying system's bandwidth requirements (such as Karaoke, Stadium, etc.). In contrast, Groove's oblivious messaging protocols allow communication with many buddies while reducing clients' bandwidth costs by orders of magnitude (see §8.2).

9 Conclusion

Groove removes the rigid requirements that prior metadata-private messaging systems imposed on clients. Groove allows users to have asynchronous text message chats with multiple buddies, while seamlessly switching between resource-constrained mobile devices. It does so with similar scalability and privacy guarantees as prior rigid differentially-private messaging systems. Groove achieves this advancement by introducing protocols for oblivious delegation that allow users to have an untrusted service provider participate in the rigid messaging protocol on their behalf. Our evaluation of a prototype of Groove shows that it can support a large user base with latency on the order of a minute. Our experiments with a Pixel 4 smartphone demonstrate that Groove can accommodate the network and power constraints of a mobile device, unlike previous rigid systems. Groove's techniques narrow the gap between metadata-private messaging and standard messaging apps, allowing for broader adoption.

Acknowledgments

The authors thank Gil Segev and Bryan Ford for helpful discussions, and our shepherd, Natacha Crooks. This work was supported, in part, by the Alon fellowship, the Hebrew University cybersecurity research center, and gifts from Microsoft and Google.

References

[1] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual conference, July 2021.

[2] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, pages 962–979, San Francisco, CA, May 2018.

[3] Sebastian Angel, David Lazar, and Ioanna Tzialla. What's a little leakage between friends? In *Proceedings of the 2018 ACM Workshop on Privacy in the Electronic Society*, pages 104–108, Toronto, Canada, October 2018.

[4] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 551–569, Savannah, GA, November 2016.

[5] Alex Biryukov, Ivan Pustogarov, Fabrice Thill, and Ralf-Philipp Weinmann. Content and popularity analysis of Tor hidden services. In *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 188–193, Madrid, Spain, June–July 2014.

[6] Circl - bls12-381, 2021. github.com/cloudflare/circl/ecc/bls12381.

[7] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, February 1998.

[8] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF)*, pages 164–178, Lisbon, Portugal, June–July 2016.

[9] Roger Dingledine and Nick Mathewson. Anonymity loves company: Usability and the network effect. In *Proceedings of the 5th Workshop on the Economics of Information Security (WEIS)*, Cambridge, United Kingdom, June 2006.

[10] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[11] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *Proceedings of the 30th USENIX Security Symposium*, Vancouver, Canada, August 2021.

[12] Philippe Golle and Ari Juels. Parallel mixing. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 220–226, Washington, DC, October 2004.

[13] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *Proceedings of the 20th Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 233–253, Kaohsiung, Taiwan, December 2014.

[14] Goran Kalic, Iva Bojic, and Mario Kusek. Energy consumption in Android phones when using wireless communication technologies. In *Proceedings of the 35th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 754–759, May 2012.

[15] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 406–422, Shanghai, China, October 2017.

[16] Albert Kwon, David Lu, and Srinivas Devadas. XRD: Scalable messaging system with cryptographic privacy. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, February 2020.

[17] Adam Langley. Pond, 2016. https://github.com/agl/pond.

[18] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[19] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 711–726, Carlsbad, CA, October 2018.

[20] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: Strong metadata security for voice calls. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 211–224, Huntsville, Ontario, Canada, October 2019.

[21] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 571–586, Savannah, GA, November 2016.

[22] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In *Proceedings of the 24th USENIX Security Symposium*, pages 383–398, Washington, DC, August 2015.

[23] Femi Olumofin and Ian Goldberg. Privacy-preserving queries over relational databases. In *Proceedings of the 10th Privacy Enhancing Technologies Symposium*, Berlin, Germany, July 2010.

[24] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *Proceedings of the 26th USENIX Security Symposium*, pages 1199–1216, Vancouver, Canada, August 2017.

[25] David Schatz, Michael Rossberg, and Guenter Schaefer. Hydra: Practical metadata security for contact discovery, messaging, and dialing. In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP)*, February 2021.

[26] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–440, Shanghai, China, October 2017.

[27] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 137–152, Monterey, CA, October 2015.

UPGRADVISOR: Early Adopting Dependency Updates Using Hybrid Program Analysis and Hardware Tracing

Yaniv David[1], Xudong Sun*[2], Raphael J Sofaer[1],
Aditya Senthilnathan[3], Junfeng Yang[1], Zhiqiang Zuo*[2], Guoqing Harry Xu[4], Jason Nieh[1] and Ronghui Gu†[1]

[1]Columbia University, [2]Nanjing University, [3]IIT, Delhi, [4]UCLA

Abstract

Applications often have fast-paced release schedules, but adoption of software dependency updates can lag by years, leaving applications susceptible to security risks and unexpected breakage. To address this problem, we present UPGRADVISOR, a system that reduces developer effort in evaluating dependency updates and can, in many cases, automatically determine which updates are backward-compatible versus API-breaking. UPGRADVISOR introduces a novel co-designed static analysis and dynamic tracing mechanism to gauge the scope and effect of dependency updates on an application. Static analysis prunes changes irrelevant to an application and clusters relevant ones into *targets*. Dynamic tracing needs to focus only on whether targets affect an application, making it fast and accurate. UPGRADVISOR handles dynamic interpreted languages and introduces call graph over-approximation to account for their lack of type information and selective hardware tracing to capture program execution while ignoring interpreter machinery.

We have implemented UPGRADVISOR for Python and evaluated it on 172 dependency updates previously blocked from being adopted in widely-used open-source software, including `Django`, `aws-cli`, `tfx`, and `Celery`. UPGRADVISOR automatically determined that 56% of dependencies were safe to update and reduced by more than an order of magnitude the number of code changes that needed to be considered by dynamic tracing. Evaluating UPGRADVISOR's tracer in a production-like environment incurred only 3% overhead on average, making it fast enough to deploy in practice. We submitted safe updates that were previously blocked as pull requests for nine projects, and their developers have already merged most of them.

1 Introduction

Powered by agile development methodologies and supported by continuous integration and testing infrastructure, modern software companies achieve blazing fast release cycles, quickly pushing bug fixes and new features to production servers or client devices. For instance, Google's Chrome ships a new major version to the stable channel every four weeks [3], while Facebook publishes updates to their front-end three times a day and releases a new version for iOS and Android every week [7].

A key enabler to this fast development cycle is the large collection of preexisting frameworks and libraries to build on. One open source software (OSS) discovery service tracking popular libraries in leading package managers lists almost 5 million open-source libraries [40]. We surveyed OSS projects developed with prominent interpreted languages[1] (§2) and found that an application, on average, depends on tens to hundreds of frameworks and libraries; these are known as dependencies.

Unfortunately, our survey shows that despite the fast pace of application updates, the adoption of dependency updates is delayed by years, and this delay is getting worse (see Fig. 1 in §2). We believe a key reason behind this dichotomy is the knowledge gap between application and dependency developers. Although dependency developers invest significant effort in creating robust and often backward-compatible updates, they typically have no direct access to the dependent applications, hindering their ability to gauge potential update risks. Application developers want the security fixes and performance enhancements in dependency updates, but lack knowledge of the dependency internals and therefore fear that dependency updates may cause the application to malfunction.

The effect of dependency update delays aggregates across projects and even whole software ecosystems. For a given installation composed of an ensemble of software components, even if only one component requires an older version of a dependency, the entire installation is forced to use the same older version. This older version might accumulate unpatched vulnerabilities over time or break unexpectedly due to deprecation. Moreover, when many older dependency versions are involved, attempts to update subsets of the dependency graph become impossible due to dependency

*Also with State Key Laboratory for Novel Software Technology.
†Also Founder of CertiK with an equity interest.

[1]We surveyed Python, JavaScript, and Ruby projects from GitHub.

conflicts (a.k.a "dependency hell" [30]).

Ideally, an application's test suite should discover any malfunctions due to interactions with dependencies, but this is sadly not the reality. Application and dependency developers strive to make their unit, integration, and system tests have high coverage of their projects. However, state of the art tools for coverage metrics do not examine the difficult-to-measure interfaces between applications and their dependencies. Thus, it is dangerous to rely on application test suites to detect dependency update incompatibilities. The problem is worse for dynamic interpreted languages, as without compilation, API breaking changes not discovered during testing become runtime errors on production servers.

We present UPGRADVISOR, a system for maximizing the safety of and reducing developer efforts invested in dependency updates. UPGRADVISOR is based on the observation that changed dependency code that does not run cannot affect application semantics. UPGRADVISOR works by combining sound static analysis with efficient dynamic tracing to aid developers in the timely adoption of dependency updates. Given a dependency update that developers want to adopt, UPGRADVISOR computes the code difference between its old and new versions and then employs static analysis to discard semantically irrelevant differences and cluster potentially meaningful ones into tracing targets.

To enable this process for modern applications written in widely-used interpreted languages, UPGRADVISOR first builds an over-approximating call graph that accurately accounts for the lack of type information in variables and function arguments in these languages as well as handling implicit language-specific call-site creation features. It then creates a fused abstract syntax tree (AST) representing both the old and new versions of the dependency and tags all changes on a per statement basis. The change tags are propagated up the AST to the call graph, clustering code differences into *call targets* (Python functions or methods) for later tracing. UPGRADVISOR can then statically discard unreachable or semantically irrelevant code changes, such as backward-compatible changes to API signatures and changes in imports location (see §7). If there are no call targets tagged with change tags, the dependency update is safe because it has no changes that can possible affect application execution. Unlike test suites, the static analysis provides complete code coverage, allowing UPGRADVISOR to accurately determine if a dependency update is safe.

While static analysis may be sufficient in many cases to determine the safety of a dependency update, it is conservative, identifying calls not actually used in practice. UPGRADVISOR therefore performs dynamic tracing to determine if call targets with change tags remaining after static analysis actually influence application execution. Dynamic tracing is performed without applying the dependency update and is designed to incur little overhead. Both of these features allow it to be used in a production environment, giving a complete trace of a production server over a substantial amount of time to serve as the ground truth of application-dependency interactions. Running UPGRADVISOR on production servers allows mitigating the inherent unsoundness of dynamic analysis.

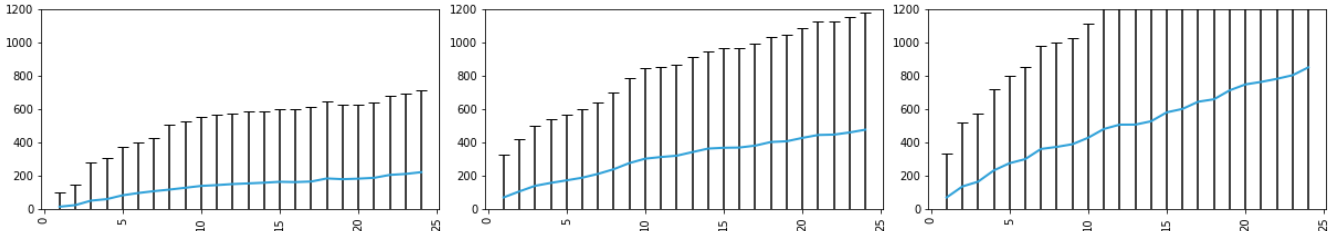
UPGRADVISOR achieves low-overhead tracing using two key mechanisms. First, UPGRADVISOR can select which parts of application execution to trace, tracing only the call targets with change tags identified through static analysis. Second, UPGRADVISOR leverages the hardware tracing module in modern CPUs using a novel coarse-grained tracing technique to collect data only for chosen bytecodes while ignoring unnecessary low-level interpreter instructions. In particular, using our technique, each bytecode branch executes exactly one native branch. Tracing only one branch creates one trace record, reducing tracing data size and runtime overhead. Combining the two allows lowering overhead while retaining precision: we only collect the minimal information required to fully capture control flow in the updated parts of the dependency.

We have built an UPGRADVISOR prototype that supports dependency updates for Python programs. It contains an analysis framework and a tracer implanted into our fork of the Python 3.7 interpreter. We evaluated UPGRADVISOR on 172 potential dependency updates that were previously blocked from being adopted by applications in top-starred OSS repositories on GitHub. The dependency updates include popular frameworks such as `Django`, `aws-cli`, `tfx` and `Celery`. Our results show that UPGRADVISOR is effective. UPGRADVISOR determines through static analysis that 98 (56%) dependencies can be automatically updated, meaning the majority of blocked dependency updates can be adopted without manual inspection. When static analysis cannot completely determine if an update is safe, the analysis reduces the code differences that must still be reviewed by an order of magnitude compared to the overall changes between old and new dependency versions. UPGRADVISOR determines through dynamic tracing that various dependencies not automatically deemed safe through static analysis can still be updated. (see §7.1)

We randomly sampled several dependency updates deemed safe: although we were not developers of either the applications or the dependencies, we were able to quickly submit pull request (PR)s, most of which were subsequently merged by the corresponding developers. The PRs that were merged included dependency updates deemed safe by just static analysis as well as by combining static and dynamic analysis, demonstrating that dynamic tracing can indeed provide additional upgrade opportunities beyond static analysis.

Finally, we performed an extensive performance evaluation, including running a production Django workload published by Instagram and Intel [8]. Our measurements show that our tracer incurs an average overhead of 3%, much lower than other tools.

UPGRADVISOR's code, evaluation datasets, and other resources are available at `http://upgradvisor.github.io`.



(a) Python (b) JavaScript (c) Ruby

Figure 1: The average delay days for all the projects surveyed for each month between August 2019 and November 2021.

2 Survey of Dependency Usage in OSS

Modern applications declare their dependency requirements in metadata files as a list of `(package name, version specifier)` tuples. These direct dependencies also have their own dependencies, creating a graph of transitive dependencies for the application. The version specifier follows a common version structure, "MAJOR.MINOR.PATCH", where MAJOR increments signal API breakage, MINOR increments signal backwards-compatible feature additions, and PATCH increments signal backwards-compatible bug fixes. For example, if the old API is `foo(int a, int b)` and the new one is `foo(int a, int b, int c)` the API was broken. A version specifier in a metadata file can be expressed as conditions, which can directly point to a specific version, also called pinning, or use a combination of lower/upper-bounding terms to define a range of possible versions. Out of a range of allowed versions, the latest one is selected. A given dependency may be specified by the application and by any number of dependencies. All these specifiers must overlap to have a viable dependency set. Using range-defining conditions allow developers to block a version update if they deem it not compatible with their code, e.g., `<=2.5.1`.

		Dependencies					
Language	Projects	Direct			Transitive		
		max	avg	std	max	avg	std
Python	389	118	7.1	11.5	480	15.9	41.3
JS	462	130	17.1	23.3	>1000		
Ruby	501	91	12.3	17.3	548	28.1	103.9

Table 1: Dependency usage in OSS projects on GitHub.

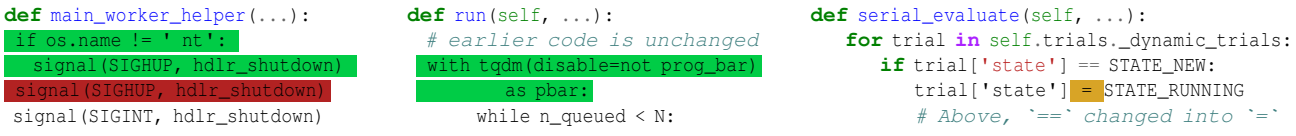
To better understand dependency usage patterns in the leading dynamic interpreted languages, Python, Ruby, and JavaScript (JS), we performed a survey of OSS projects using them. We randomly sampled top starred (>1k) project repositories on GitHub, for which Python, Ruby, or JS, was the primary programming language. Starting from 1,382 Python, 913 Ruby, and 1,144 JS projects, we examined the dependency requirement conditions of the latest version of each project and filtered those with no direct dependents as of November 2021.[2] Table 1 summarizes the results for projects with dependencies, showing the maximum, average, and standard deviation in the number of dependencies per project. Each project is considered an application. For example, Python applications averaged seven direct dependencies and 16 transitive ones for an average of 23 total dependencies per application, but the standard deviations (STDs) show significant differences among applications. The number of direct and transitive dependencies for a Python application was as high as 118 and 480, respectively.

For the 389 Python applications, we examined the dependency requirement conditions for not just the latest version of the application, but also earlier versions published from August 2019 to October 2021. 2% have no restrictions (latest), 29% are lower-bound only, 38% are double-bound (both lower- and upper-bound), and 31% are pinned version specifiers. In other words, more than two-thirds of the version specifiers, double-bound and pinned, may block available updates. A developer whose application may have dozens of dependencies, including transitive dependencies, cannot update dependency X unless every other dependency which depends on X also includes the new version in the specifier.

We measured the historical delay for Python, Ruby, and JS applications in updating their dependencies by examining all versions of the applications published from August 2019 to November 2021. For each released application version, we examine direct dependency requirements. Considering only the dependency versions which existed on the application version's release date, we check if the dependency offered an updated version. If an updated version exists, we consider the application to be delaying updates and measure the number of delay days. Delay days are counted from the dependency's new version release date up to the application's release date. If an application has multiple delayed dependencies, we consider only the most severely delayed dependency.

Fig. 1 shows the delay days for all applications as measured each month from August 2019 to November 2021. We show both the average delay days as well as the standard deviation. For example, Fig. 1a shows that Python applications start from

[2] Unlike JS and Ruby, Python projects declare dependencies implicitly in their setup scripts. We discarded projects when we could not extract dependency constraints.



(a) (b) (c)

Figure 2: Three code change snippets from `hyperopt`'s update from version 0.1.1 to version 0.1.2.

an average of roughly 20 delays days for August 2019 and balloon to reach roughly 200 delay days by August 2021, an order of magnitude increase in delay over two years. Fig. 1 shows that this pattern of increasing delay in adopting dependency updates persists across applications in all languages, indicating that the problem of timely adoption of dependency updates worsens over time. Digging into the data shows that while some projects invest consistently in dependency upkeep, other projects struggle. This difference leads to the significant variations as expressed by the standard deviation bars in Fig. 1. The standard deviation in delay days is so large for Ruby applications that they exceed the visible range in Fig. 1c for most months; the visible maximum was capped at 1,200 delay days to provide a consistent visual comparison across languages while keeping the graphs readable. Because dependency requirements cater to the lowest-common-denominator, having even one such struggling project as a dependency forces the use of an old version.

We designed UPGRADVISOR to address this problem.

3 UPGRADVISOR Overview

We use `Qlib`, a popular Python AI-oriented quantitative investment platform developed by Microsoft, as a motivating example of the dependency update problem and show how UPGRADVISOR solves it. `Qlib` version 0.7.1, released on 15-Sep-2021, relies on 30 direct dependencies. One of them is `hyperopt` 0.1.1, released on 27-Aug-2018, a distributed asynchronous hyper-parameter optimization library for Python.

3.1 An Example Dependency Update Problem

`hyperopt`'s developers changed 828 line of code (LOC) spanning 14 files to go from version 0.1.1 to 0.1.2. Because `Qlib` uses a pinned version specifier "hyperopt==0.1.1", it did not adopt version 0.1.2. Counting the days between `hyperopt`'s version 0.1.2 release on 21-Feb-2019 to `Qlib`'s 0.7.1 release on 15-Sep-2021, the number of delay days for `Qlib` due to not updating `hyperopt` is 937.

To update `Qlib` to use `hyperopt` version 0.1.2, `Qlib`'s developers need to ensure the update is safe. It should not cause `Qlib` to crash, experience other silent failures, or change `Qlib`'s API. A change in `hyperopt`'s output content or structure could propagate to `Qlib`'s output. An update solving a bug in `hyperopt` might benefit `Qlib`, yet still requires `Qlib`'s developers to check for unexpected side effects. Ideally, `Qlib`'s developers can use the opportunity of updating to a new `hyperopt` version to incorporate improvements in `hyperopt`'s functionality they already use or explore its new features.

This process offers the developers a tradeoff between short-term safety by not updating versus investing efforts towards gaining long-term safety and quality. We aim to maximize the safety of the update and its benefits while reducing the developer's efforts required to examine the dependency update.

The easiest way to evaluate the updated dependency is to run `Qlib`'s test suite with `hyperopt`'s new version. It turns out that all of `Qlib`'s tests pass. Sadly, this result is ambiguous as it can not differentiate between the tests not covering `hyperopt` and the update being safe. In fact, measuring the coverage of `hyperopt` when running `Qlib`'s test suites shows that no line in `hyperopt`'s code is covered.

Instead of using its test suite, `Qlib`'s developers can examine all code changes made to `hyperopt` to assess the safety of the update. Fig. 2 shows a few changed code snippets from `hyperopt`'s update. Fig. 2a shows a change in the way worker helpers initialize signal handlers. After the update, when the code runs in a Windows environment, the `SIGHUP` handler is no longer set. Fig. 2b shows a change in the `run` method in charge of running the trial's computations, adding an optional progress bar (controlled by the `disable` flag). Fig. 2c shows a change to the `serial_evaluation` method, turning the condition on `trial['state']`, which was never assigned (effectively redundant code), into an assignment.

Fig. 2a and Fig. 2c are bug fixes, which might benefit `Qlib`, but Fig. 2b constitutes a change to `Qlib`'s CLI, which might break other systems using the CLI output.

The changes in these procedures[3] require human review. Doing this for a few changes may be manageable, but is too difficult for all changes on each update; manual code examination is not scalable.

3.2 Using UPGRADVISOR to Update `Qlib`

UPGRADVISOR is based on the observation that changed dependency code that does not run cannot affect application semantics. If we can show such code is unreachable, we can ignore it. Fig. 3 shows UPGRADVISOR's process for analyzing the dependency update (steps 1-3), employing the tracer (steps 4-6), and gathering and summarising results towards update advice (steps 7-8).

[3] For brevity, we use procedure in place of "method or function".

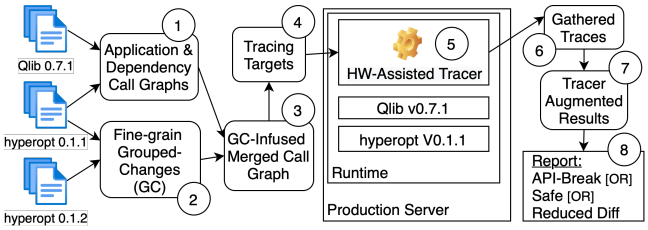


Figure 3: UPGRADVISOR's process of analyzing, tracing and providing update advice for our motivating example.

Analyzing the dependency update. Our analysis goal is to determine statically if the update is safe, or if not possible, reduce the number of changed procedures that need to be tracked by the tracer or examined by a developer. Static analysis involves the following steps.

Step 1: Build call graphs for `Qlib` (the application) and `hyperopt`'s old version (the dependency). Call graph nodes represent procedures, and directed edges represent call relations.

Step 2: Compare `hyperopt` versions to create a fused abstract syntax tree (AST) containing a set of fine-grain change tags. Change tags label the affected AST subtree with the type of change made and the change position in the source code. As we see later, specific change types and location combinations will be handled differently. A change can be a statement modification (e.g., Fig. 2c), an addition of several statements to an existing procedure (e.g., Fig. 2a), or the deletion of a method from a class, possibly breaking any code calling it. We group all change tags in the same procedure because UPGRADVISOR traces at procedural level.

All non-semantic changes, adding a space or changing comment text, are ignored by using an AST representation. Change tags are discarded by employing a language-specific analysis using the AST-subtrees content. For example, for Python, any changes involving type annotations and order changes between unrelated import statements are discarded.

Step 3: Merge the application and dependency graphs, connecting all interfaces between `Qlib` and `hyperopt` in the graph, and infuse the grouped changes into the relevant graph nodes. We discard `hyperopt` nodes that are not reachable from any of `Qlib`'s nodes, along with any change tags connected to these nodes. For example, the changes depicted in Fig. 2a are discarded as no graph path from `Qlib` into `hyperopt` leads to the `main_worker_helper` function.

Starting from 72 changed procedures in `hyperopt`, performing steps 1-3 leaves only four nodes with change tags in the merged graph. Fig. 4 shows part of the merged call graph containing these four nodes, which represent changed procedures. `Qlib`'s only procedure calling into a changed procedure in `hyperopt` is `contrib.tuner.(...)` (in green), calling `fmin`, a part of `hyperopt`'s API (in orange). The changed procedures, e.g., `FMinIter.init` are marked as red stars, while other non-changed `hyperopt` procedures connecting them, e.g., `(...).exhaust`, are shown as well (in red).

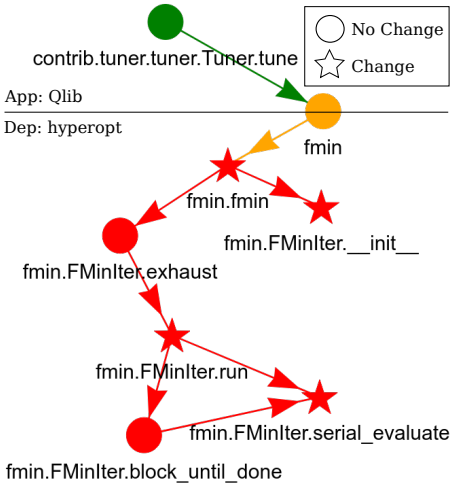


Figure 4: The graph of `hyperopt`'s code changes reduced to only show changes affecting `Qlib`.

Employing the tracer. UPGRADVISOR traces the existing dependency code, ideally running on a production server. These traces can then be used to simulate the dependency update, which can catch breaking changes and discard changes to unreached parts of the dependency.

Step 4: After statically determining the four changed `hyperopt` procedures which might be reachable from `Qlib`'s code, their names are sent on the fly to the tracer already running on the production server.

Step 5: The tracer then starts tracking them by logging every control-flow decision in the procedure, including conditional branches and exceptions. `Qlib`'s test suite did not cover any of `hyperopt`'s code and specifically did not exercise `contrib.tuner.(...)` which calls the changed part of `hyperopt` from `Qlib`. However, if a production environment is not available to trace, the static analysis provides insight on what kind of test cases should be created to provide better coverage. For this example, we manually created a production-like workload which covered calls to `hyperopt` and ran them on the traced system. Specifically, we made `Qlib` use `hyperopt`'s asynchronous computation mode. Tracing relevant methods in `hyperopt` incurs only ~5% runtime overhead on the system.

Step 6: The tracer's output is decoded offline to reconstruct execution traces for tracked procedures.

Gathering and summarizing results. Step 7: The graph created in step 3 is augmented with the collected traces. Any change tag is excluded if its code location is not present in the traces. If all tags in a group are excluded, the whole procedure is discarded. The changed statement in `serial_evaluate`, shown in Fig. 2c, does not exist in the traces, so it is discarded.

At this stage, only three changed procedures, including `run` shown in Fig. 2c, require manual examination. Taking a closer look at the changes in these three methods shows

that all changes relate to the addition of the progress bar in `run`. `fmin.fmin`'s signatures adds a new variable: `def fmin(... , prog_bar=True): ...`. Its value is then propagated to the `fmin.FminIter` class, which then uses it when calling `tqdm(disable=prog_bar)`.

Adding arguments to a function declaration might be a source for API breakage, as a change to required positional arguments might cause a runtime error. In this example, because the new argument has a default value ("True"), a runtime error will not occur. UPGRADVISOR still marks this update as a "possible API break" due to the change in `Qlib`'s output caused by the progress bar. Specifically, this can be avoided by changing `Qlib`'s code to assign "False" to `prog_bar` when calling `fmin.fmin`. Following this, developers can move forward with the updating `hyperopt` to version v0.1.2.

We submitted a PR to the `Qlib` project, recommending the changes described above. This PR was adopted quickly by the maintainers and merged into the `Qlib`'s main code branch within five hours, even though `hyperopt`'s version 0.1.2 had been available almost three years (released 21-Feb-2019) at the time of the PR.

4 Static Analysis of Dependency Updates

As shown in steps 1-3 from Fig. 3, UPGRADVISOR uses static analysis to determine if a dependency update is safe, or identify what procedures may be affected by the update so they can be further considered by dynamic tracing. The inputs are the application code, A, and dependency code D in two versions before and after the update, D_{Before} and D_{After}, respectively.

Throughout this section, we use Python terminology for methods and functions, where a method is a block of code associated with a class and a function is a block of code that can be called but is not associated with a class.

4.1 Application and Dependency Call Graphs

UPGRADVISOR first builds call graphs for A and D_{Before}, which are merged into one graph G. Building an accurate call graph requires: (1) mapping call sites and (2) detecting callees (call targets). However, dynamic interpreted languages such as Python typically do not require specifying types, causing callee uncertainty. Consider the following Python snippet:

```
def foo(a):
    return a.get_size()
```

The function `foo` has an untyped argument `a`, and it calls `a`'s method `get_size`. `a` can be any class that has a method `get_size`, and there is no type information to help narrow down the potential callees. We refer to `get_size` as a named method with an unknown class because the method name called is known but the class to which it belongs to is unknown. Alternatively, consider the following Python code snippet:

```
def foo(a):
    return a()
```

The function `foo` has an untyped argument `a`, and it calls `a`. `a` can resolve to any function in the code, and there is no type information to help narrow down the potential callees. We refer to `a` as an anonymous function. There are ways to explicitly specify types in Python using type annotations [36], as in the following code snippet:

```
def foo(a:arg_type) -> ret_type:
    return a.get_size()
```

However, this is optional in Python, so call graph construction must account for the absence of types.

We use call graphs to decide if an update is safe or identify tracing targets, so their soundness is crucial. While false edges can be tolerated (false positives), there cannot be missing edges (false negatives). We achieve this by over-approximating calls in the graph. The basic idea is to use type information when available to build a context-sensitive [15] call graph to pinpoint the exact method called, but then combine this with context-insensitive analysis for missing targets. We split missing targets into two types: (1) named methods with unknown class and (2) anonymous functions. To express the first type of missing targets in our call graph, we create an edge with a "magic" prefix followed by the callee name, e.g., `UNK.get_size`. To express the second type in the graph, we create a magic edge from the node to `ANON`.

Using the process described above, we construct call graphs for A and D_{Before}, and merge them into one graph $G = (V,E)$ with V nodes and E edges. We split V into two groups depicting M methods or F functions, respectively: $V = M \cup F$. To make G over-approximate for missing call targets we apply the following edge adding rules:

1. $(n,UNK.x) \in E, \exists y.x \in M \Rightarrow E = E \cup \{(n,y.x)\}$
2. $(n,ANON) \in E, x \in F \Rightarrow E = E \cup \{(n,x)\}$.

The first rule adds edges from the respective node to all methods with the same name as the named method with unknown class. The second rule adds edges from the respective node to all functions. These rules add all possible call targets for named and anonymous missing targets. Exploring the Python projects discussed in §2, we find a limited amount of named method missing targets exist in almost every project, while anonymous function missing targets were scarce.

Due to their scripting-oriented roots, most dynamic languages allow placing statements in the source-code file outside of procedures or classes. Running this file as a script or importing it from another file will execute these statements. For example, given a file named "h.py" including `print("Hello World")`, putting the import statement `from h import *` in another file will result in "Hello World" printed on the screen. To represent these statements in the call graph, we place them into a special `module_ctor` pseudo-procedure node and add an edge to relevant importing files.

A graph will contain an edge from a procedure to `module_ctor` if the procedure contains the relevant import statement.

Similarly, we place class fields and their optional initialization in a special pseudo-class-initializer `X_cinit` node, adding edges to and from it between every call site to any class constructor. For example, a statement creating a new class instance, `ClassA()`, placed inside a procedure named `foo` will create the following call path: `foo` → `ClassA_cinit` → `ClassA_ctor`.

We treat other language-specific container for representing code, such as Python's decorators (see §6), similarly.

4.2 Grouping Changes

UPGRADVISOR introduces a novel static approach for creating grouped fine-grain changes. We introduce *change tags*, used for tagging individual statements that have changed between D_{BEFORE} and D_{AFTER} as additions, deletions, or modifications. These fine-grain per statement tags are then grouped together by the lowest-level procedure that contains the respective tags.

UPGRADVISOR fuses the code in D_{BEFORE} and D_{AFTER} into one AST and marks changes with change tags. For example, for Python, we create one AST per Python module. Each module is contained in a file and has procedures, classes, and other statements. The fused AST contains all deleted and added statements, while modified statements contain the code from D_{BEFORE}. For modified code, the code in D_{BEFORE}'s copy is stored in the AST because UPGRADVISOR will later need to identify D_{BEFORE} code when combining it with collected traces generated by running D_{BEFORE}, as discussed in §5. Each change tag represents a change in a statement and contains a pointer and a type, the type being either addition, deletion, or modification. The pointer points to the affected statement, i.e., the lowest statement-tree-node containing the change. For example, in Fig. 2c, the modification tag is applied to AST node representing `trial['state'] = ...`, while in Fig. 2b an addition tag is applied to the node representing `with tqdm(...)`, and no tag is applied to the node representing `while n_queued`. Changes to procedure declarations, such as adding an argument or default value for one, are represented as a tag on the procedure's declaration node in the AST. If a file was deleted or added, we create an AST with all statements and procedure declarations containing deletion or addition tags to represent it. Change tags are then grouped by the lowest procedure, class, or module containing them by following each AST pointer and moving up the tree.

4.3 Clustering Changes Into Call Targets

UPGRADVISOR attaches the grouped changes to nodes in the call graph G, discussed in §4.1. As grouped changes are associated with the lowest procedure, class, or module containing them, it is straightforward to attach them to nodes in the call graph. Any node with at least one change tag attached to it is considered a changed node. Note that changed nodes exclusively appear in the part of G constructed from D_{BEFORE}.

UPGRADVISOR then performs the following two steps. First, it discards change tags that, in G's context, do not affect the semantics of the code. Examples include (1) called APIs adding unused default values, and (2) changes in import location or procedures moving between files. If all change tags in a specific group were discarded, the node associated with this group is no longer considered a changed node. Second, UPGRADVISOR discards any changed node not reachable from an application node. Any changed nodes remaining after this two-step process are marked as call targets, and their corresponding procedures will then be sent to the tracer. These call targets represent changes that can potentially affect the application. If there are no call targets, static analysis alone was successful in automatically determining that the update is safe.

Propagating the indirect effects of direct updates to data is currently out of scope for UPGRADVISOR. These include direct updates to external data used by the code, such as HTML templates, or changes to data in the code itself, such as data used for initialization. UPGRADVISOR can be configured to report on changes to external data. As changes to data in the code are necessarily a changes to the code, these will be detected statically through the call graph if it is reachable from the application, ensuring the correctness of the static analysis. However, any effects due to changed data on other non-changed parts of the code will not be propogated. For example, if a dependency's internal state, such as a global variable, is updated and an unchanged method reads this global variable, UPGRADVISOR will not identify the unchanged method as a call target. UPGRADVISOR can be expanded to propagate the effect of the changed state and mark these methods for tracing or report more methods for developer inspection, and we intend to explore this in future work. As discussed in §7, we find such transitive state changes in the code to be rare.

5 Dynamic Hardware Tracing

UPGRADVISOR uses dynamic tracing to determine what an application actually does in practice. By tracing application execution in a production environment, we can obtain the ground truth of application-dependency interactions and see which call targets are actually used. To allow dynamic tracing in production environments, it is crucial that tracing have minimal impact in production, including avoiding application changes and incurring minimal overhead. For the former, UPGRADVISOR traces the existing application without applying any dependency updates, so no application changes are required. For the latter, UPGRADVISOR introduces two key mechanisms, target-focused tracing and hardware-assisted coarse-grained tracing for interpreted languages.

```
// given a code block with a sequence of bytecode
for (each opcode in code block){
  switch (opcode) {
    case opcode_1:
      subroutine_1(); //interpretation logic for opcode_1
      break;
    case opcode_2:
      subroutine_2(); //interpretation logic for opcode_2
      break;
    ...
    case opcode_i:
      subroutine_i(); //interpretation logic for opcode_i
      break;
    ...
  }
}
```

Figure 5: Original interpretation loop inside an interpreter.

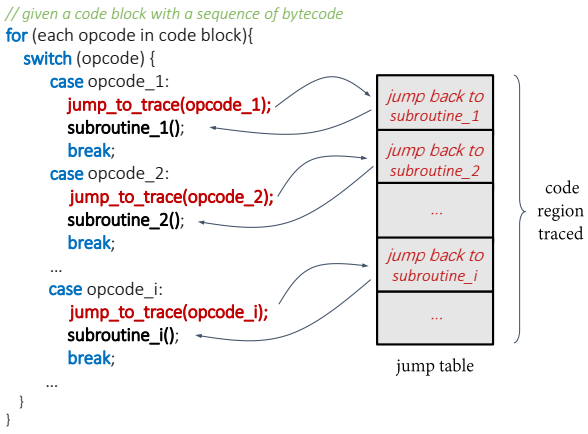


Figure 6: Traced interpretation loop inside an interpreter.

5.1 Target-focused Tracing

UPGRADVISOR does not need to trace the entire application execution, but needs only to trace call targets generated from the static analysis discussed in §4.3. This small handful of methods is not known in advance, and may change for different updates. For languages such as Python, a compiler compiles the program written in the interpreted language to a sequence of bytecode instructions, and the interpreter runs a loop that interprets bytecodes one by one at runtime, as shown in Figure 5. UPGRADVISOR enables on-the-fly selection of which methods are traced by interposing on the interpretation loop used to interpret the intermediate bytecode for dynamic interpreted languages. This logic is illustrated in Listing 1.

```
1 // given a code block with a sequence of bytecodes
2    maintain set of methods to be traced;
3    if(signature of code block is in the set)
4        { goto traced loop; }
5    else{ goto original loop; }
6    original loop:
7        loop code shown as Figure 5;
8    traced loop:
9        loop code shown as Figure 6;
```

Listing 1: UPGRADVISOR's target-focused tracing check logic.

We modify the interpreter to allow running a traced version of the loop on demand. UPGRADVISOR maintains a set, updatable during runtime, of signatures for all methods marked for tracing. Before running any method, the interpreted checks if it is part of this set, directing the execution to the traced or original version (where no tracing is enabled) of the loop accordingly. The traced loop is shown in Figure 6, which only differs from the original loop by adding a jump instruction before each call to a subroutine in the interpreter loop, which enables tracing as discussed further in §5.2.

5.2 Coarse-grained Hardware Tracing

To further reduce tracing overhead, UPGRADVISOR leverages hardware tracing mechanisms widely available in modern CPUs, specifically Intel Processor Trace (PT) [21]. Intel PT records dynamic control-flow information such as branch targets and branch taken indications, encoding them as trace packets. With the trace packets collected and the program's native code as input, a software decoder [20] can then be invoked to reconstruct the control flow of the program executed. Although hardware tracing has advantages in terms of low overhead and the absence of intrusiveness, a key challenge is how to leverage it to meaningfully trace interpreted languages since it can only profile native instructions directly running on physical CPUs [43]. For a native program, native instructions can be readily mapped back to the source code with the aid of compilation metadata. This is not the case for programs written in interpreted languages. For interpreted languages such as Python running in a virtual machine, the intermediate bytecode corresponds to the source code, but the native instructions executed by the CPU are those of the interpreter.

To leverage the efficiency of hardware tracing, we need to develop tracing support that can bridge the gap by relating hardware traces generated by CPU to bytecode instructions of interpreted languages that developers can understand. A naive way to obtain the execution trace at the bytecode level is to trace the execution of the entire interpreter code and then reconstruct the execution flow of high-level bytecode based on the mapping between bytecode types and their respective interpreter subroutines. For example, Intel PT generates trace packets with instruction pointers (IPs) to identify the address range for each instruction. In Figure 5, interpreter subroutines such as `subrountine_1` and `subrountine_2` have static address ranges for their instructions. Knowing that the executed instructions are within the address range of a particular function suggests which bytecode opcode is being interpreted.

Unfortunately, this approach may suffer from data loss as it can record a huge amount of unnecessary low-level trace data.

Intel PT uses a memory buffer to store trace data. Data loss occurs when there is more trace data generated than can be written into the buffer. It is extremely challenging to determine after the fact what data is lost and how to recover it [43]. However, what we are interested in is only the sequence of bytecode instructions executed, not the low-level control flow of the interpreter subroutines. What is needed is a *coarse-grained* tracing mechanism that focuses on the collection of the high-level bytecode sequences without capturing extraneous details of the subroutine implementations.

To this end, we developed a novel coarse-grained tracing mechanism that avoids capturing low-level interpretation instructions. We leverage a feature of Intel PT that allows trace packets to be filtered based on their IPs. An address range can be specified such that packets whose IPs are not in the range will be filtered out by the CPU. We create a trampoline (i.e., *jump table*) and use it as a special memory region that allows us to quickly filter out irrelevant instructions while retaining those that correspond to the bytecode. As shown in Figure 6, the jump table consists of a sequence of contiguously allocated *tablets*, each corresponding to a particular opcode. A tablet contains only one single jump instruction that jumps back to the call to the subroutine for the opcode. The traced interpretation loop has a jump instruction before each call to a subroutine in the interpreter. This instruction takes the control to its corresponding tablet; executing the instruction in the tablet takes the control back to the interpreter code. Essentially, the interpreter takes a "detour" to visit a specific (a priori known) address range defined by the jump table. We use this address range to allow Intel PT to filter out all instructions whose IPs are not in the range. As a result, the trace that PT ends up generating contains only the executed jump instructions in the tablets, and these instructions immediately reveal the bytecode opcodes due to their one-to-one mapping.

5.3 Gather Trace Results

Once hardware traces are collected, we decode them offline to reconstruct the dynamic control-flow of the program execution and deduce the code executed at runtime. The decoder decompresses the hardware trace data as a sequence of executed jump instructions, each corresponding to one tablet in the jump table. Using the one-to-one mapping between tablets and bytecodes, we reconstruct a partial sequence of bytecodes interpreted at runtime. Using the static control-flow graph for each traced method and partial bytecode sequence we project the sequence of bytecodes onto the graph so as to reconstruct the dynamic control flow executed. Once the concrete dynamic control-flow is determined at the bytecode level, we then leverage the available compilation metadata to obtain the exact lines of source code executed.

We then return to the call graph discussed in §4.3 and discard additional changed nodes based on the trace results. Specifically, UPGRADVISOR discards any change tag not associated with a statement present in the traces. Any remaining changed nodes are used to create a reduced diff file, containing differences between D_{BEFORE} and D_{AFTER} where only reachable changes appear. This reduced diff file is then made available to the developer for further examination to determine if the update is safe for adoption. If there are no changed nodes remaining, the update is considered safe.

The current version of UPGRADVISOR lacks support for exceptions. Once an exception is raised, the exception mechanism's unusual execution flow affects the control-flow reconstruction mentioned earlier. In future work, we would like to support exception handling. In brief, an exception redirects execution to a dedicated block inside the interpreter. This block is responsible for directing the execution flow back to the corresponding exception handling bytecode determined by the point where the exception occurs. Supporting hardware tracing of exceptions requires tracing that redirection block to bridge the exception control flow gap.

Apart from interpretation, certain language runtimes also enable just-in-time (JIT) compilation mode for the sake of performance. Our design focuses on interpreted mode. Adding a similar design to the one we proposed by [43] will allow for hardware tracing JITed code.

6 Implementation

We have implemented an UPGRADVISOR prototype for Python 3 applications. We built the static analyzer on top of Pyre-check [9], a type-checker for Python 3. Pyre infers missing types and generates a set of calling targets for each call site it soundly resolves. For non-resolved targets, we inserted the magic edges explained in §4.1. To perform an AST-based code comparison, we used `GumTreeDiff` [10], a state-of-the-art code differencing tool employing its JSON-edit scripts creation function to help generate fused and tagged AST.

UPGRADVISOR handles Python decorators [35] by defining them as procedures so they are represented as nodes in the call graph. For example, given a function `bar` decorated with `@dec`, a function `foo` calling `bar` will result in the following graph path: `foo` → `dec` → `bar`. We leave for future work a more subtle analysis allowing separation of the different parts of the decorator logic (i.e., set up, wrapper and cleanup) and subsequent graph edge creation. Any change (add, modify or delete) to a procedure's decorator or its arguments is handled similarly to a procedure declaration change.

We built the hardware tracer on top of CPython [12], the default and most widely used interpreter of Python. In CPython, the interpretation functionality is directly written as a loop in C code and Python code is compiled into executables once the interpretation starts. We modified the interpretation loop as explained in §5. Instead of allocating a buffer, we statically inserted a trampoline block (equivalent to a jump table) into the interpreter's codebase. As CPython does not feature any JIT-related optimizations, we only need to monitor bytecodes

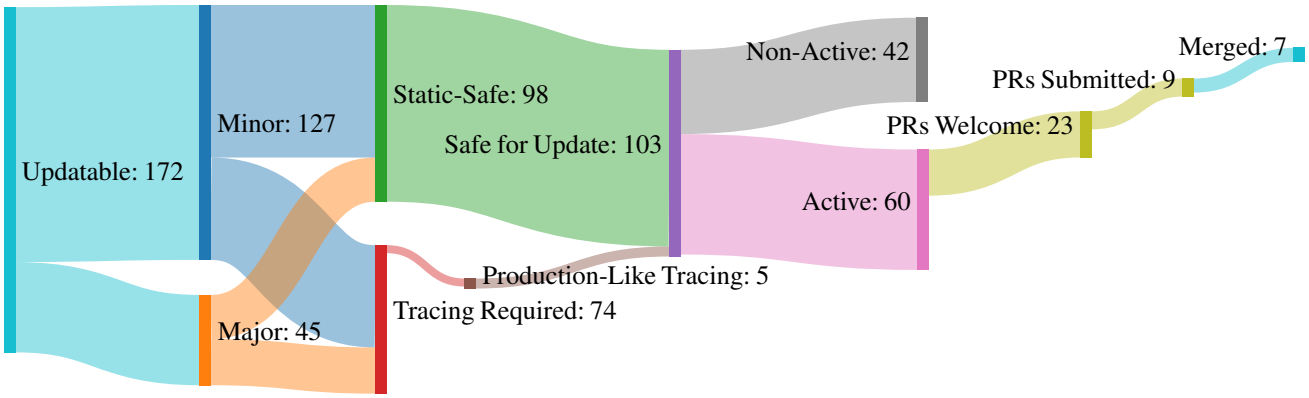


Figure 7: UPGRADVISOR's effectiveness on 172 dependency updates. Its hybrid static and dynamic analysis identified 102 updates as safe. A sample of safe updates were submitted as PRs, almost all of which have been merged.

resulting in control-flow divergence. Five kinds of Python bytecode are taken into account: *POP_JUMP_IF_FALSE*, *POP_JUMP_IF_TRUE*, *JUMP_IF_FALSE_OR_POP*, *JUMP_IF_TRUE_OR_POP*, and *FOR_ITER*. Each of them has two potential branches, true or false. Thus, the trampoline block has ten tablets.

The current prototype supports only applications written entirely in Python. Performance-minded Python projects may convert computation-heavy code into C. A prominent example is `numpy`, a scientific computing package. We leave extending UPGRADVISOR's approach to mixed language projects for future work, and currently UPGRADVISOR will alert and stop processing when C code is detected in the project. The prototype supports tracing only on bare-metal machines. To extend it to run in a virtualized environment (e.g., VMs or containers) will require OS support and further changes in memory mappings for tracing. We note that Intel already added initial support to KVM [19], and leave the rest for future work.

7 Evaluation

We evaluated the effectiveness of UPGRADVISOR in adopting blocked dependency updates and its performance overhead. We first used UPGRADVISOR to examine possible Python dependency updates from our survey discussed in §2. Although the vast majority of the 389 Python applications blocked dependency updates, we only considered those written entirely in Python 3. Altogether, we examined 50 applications with 172 possible dependency updates. We further tested UPGRADVISOR's ability to detect API breakage using known API changing updates. We then measured the performance overhead of UPGRADVISOR's tracer using a subset of the 50 applications with available performance test suites. Finally, we also measured UPGRADVISOR's tracer performance using Instagram's `django-workload` [8], based on a real-world large-scale production workload.

Static analysis was done on a machine with an AMD Opteron 6168 CPU (48 cores) and 62GB of RAM. Dynamic tracing was done on a machine with an Intel i7-10700 CPU (8 cores) with 16 GB of RAM. All machines ran Ubuntu 16.4.

7.1 Facilitating Dependency Updates

We evaluated UPGRADVISOR's ability to adopt 172 previously blocked dependency updates for 50 GitHub projects, including `Django`, `aws-cli`, `tfx` and `Celery`. Some of these projects were also dependencies for other projects. When the latest version of a project blocked a dependency update, by pinning or double-bounding dependency requirement conditions, we explored the possibility of removing the block and updating it to the next version of the dependency. For example, in our motivating example presented in §3, `Qlib` v0.7.1 pinned the dependency `hyperopt` to version v0.1.1, while version v0.1.2 exists. Out of these 172 possible updates, 45 were major version updates, and the other 127 were minor. Fig. 7 depicts the high-level view of this process.

UPGRADVISOR's static analysis was able to determine that the majority of dependency updates, 76 minor and 22 major, were safe and could be automatically updated without further dynamic tracing. These 98 updates are marked as "Static-Safe" in Fig. 7. Referring back to our survey for update delays in Python, Fig. 1a, performing all of these updates to the next available dependency version would save an aggregate of 11,310 delay days, averaging 115 delay days saved per dependency. We further confirmed the "Static-Safe" results by sampling roughly 10% of them, 11 to be exact, and manually validated that the code changes were safe.

We measured the reduction in code differences that still remained to be considered after static analysis versus the entire code differences of the updates. The total number of diff lines in all 172 updated versions we considered for this experiment was 667,604, with the average update constituting 3,881 diff lines (STD 9,078). While not a perfect metric, we use diff size as a proxy for manual developer effort required to study a de-

Project (Dependency)	Diff (LOC)	% Discarded		
		Static	Dynamic	Total
AutoML (distributed)	850	95	5	100
Electrum (qdarkstyle)	641	88	8	96
Flair (gdown)	1500	71	29	100
Qlib (Hyperopt)	828	90	9	99
Scylla (requests)	449	90	8	98

Table 2: Diff reduction for dependency updates, showing diff size in LOC and the percentage of lines discarded statically, using dynamic tracing, and in total.

pendency update. UPGRADVISOR's static analysis was able to reduce the diff sizes by an average of 91%. The reductions are consistently large across updates, with a standard deviation of 17.58%. These reductions also count cases in which UPGRADVISOR finds the update safe, eliminating the whole diff file.

We also quantified the prevalence of direct changes to data such as global variables that could potentially be used by unchanged methods. We found that only 10 out of the 172 updates contained such transitive state changes, indicating that they are infrequent. Furthermore, UPGRADVISOR was able to statically determine 5 of the 10 as safe, so only the remaining 5 still requiring dynamic tracing could be impacted by the current limitation of UPGRADVISOR not identifying unchanged methods using changed data.

Among the remaining 74 dependency updates that could not be resolved statically, denoted "Tracing required" in Fig. 7, we selected a representative sample to evaluate further using dynamic tracing. The specific projects and dependencies evaluated are listed in Table 2.

Unfortunately, we did not have access to actual production environments for these applications, so we used the results of the static analysis to help construct production-like workloads to cover application-dependency interactions for these applications. For `AutoML`, an automated machine learning framework, we ran selected sk-learn tutorials. For `Electrum`, a GUI-based Electrum Bitcoin wallet, we manually interacted with the GUI to try and trigger the relevant parts of the dependency code. For `Flair`, a framework for state-of-the-art (SOTA) Natural Language Processing (NLP), we used publicly available datasets for multiple supported languages (used for training), employed trained models, and ran tutorial examples. For `Qlib`, we set up a MongoDB instance to allow `hyperopt` to conduct asynchronous hyper-parameter optimization, and generated testing inputs for various optimization calculations, as discussed in §3.2. For `Scylla`, a proxy search and connection tool, we scanned for available proxies and used them to crawl major news sites. When applicable, to further increase coverage for possible program behaviors, we used inputs included by the project or created in our environment to drive the `atheris` fuzzer for Python [14].

Table 2 shows the results of running UPGRADVISOR end-to-end process on the project's production-like environments.

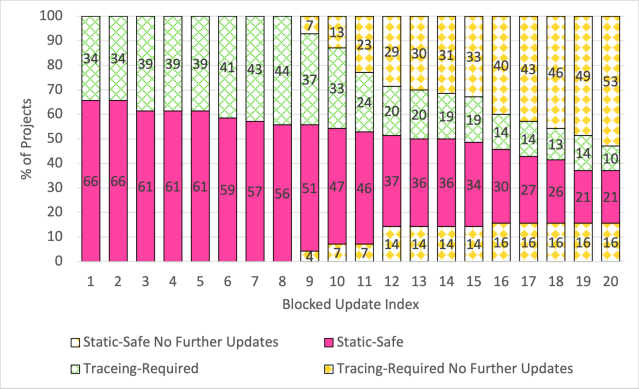


Figure 8: Using UPGRADVISOR on 75 application-dependency pairs with eight or more blocked updates.

On average, using the tracer further reduced diff sizes by 12%. Furthermore, the tracer allowed for classifying more updates as safe. For other updates, e.g., `Qlib`, additional manual inspection was required as not all code changes could be discarded from dynamic tracing, but only ~2% of the original code changes required manual inspection, significantly reducing developer effort in adopting the dependency update.

7.2 Analyzing Multiple Blocked Updates

When applications fail to perform their dependency's first update, subsequent updates are blocked as well. Among the 172 blocked dependency updates, the number of blocked updates per dependency is 12.5 on average, the median being 5, with a standard deviation of 43.67. For example, by pinning `hyperopt` to version 0.1.1, `Qlib` blocked eight updates, from 0.1.2 to 0.2.6. More generally, among the 172 blocked dependency updates, there are 75 dependencies with eight or more blocked updates.

Fig. 8 shows the result of using UPGRADVISOR on each of the eight or more blocked updates for the 75 dependencies. The blocked update index indicates how many versions after the adopted dependency is the update being considered. For example, the first bar shows the next version of the dependency, which is the subset of results from the study in §7.1 limited to just these 75 dependencies. For each blocked update index, we show the percentage of updates UPGRADVISOR requires tracing for as opposed to deeming safe statically. Starting from 34%, this percentage steadily increases to 44% in the eighth update, constituting a ~30% increase. If we count blocked updates as retaining their previous status (static-safe or tracing required) when no further updates are available, this trend continues as the blocked update index increases from 9 to 20.

To test UPGRADVISOR's hybrid approach contribution to the analysis of multiple blocked dependency updates, we employ our production-like testing environment to `Qlib`'s `hyperopt` dependency for all available updates. Fig. 9 shows diff sizes and UPGRADVISOR's ability to statically and dynamically

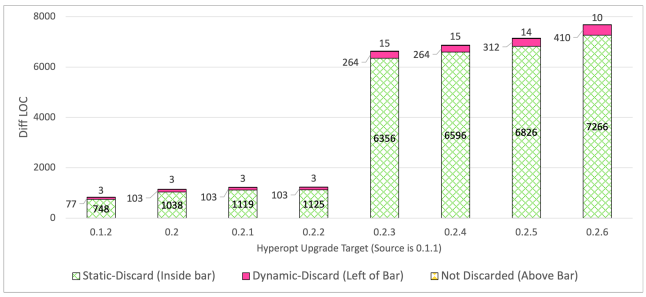


Figure 9: Diff sizes and static and dynamic discards for `hyperopt`'s eight updates.

discard changed code across `hyperopt`'s eight updates. Note that the first bar represents the same `Qlib` data as in Table 2.

7.3 Contributing to the OSS Community

To further validate our results, we selected a sample of dependency updates that UPGRADVISOR considered safe and submitted them to the respective project via a PR. Except `Qlib`, which was the first PR we submitted, all other dependencies were updated to their latest version. As submitting a PR requires manual effort, we focused on active projects welcoming PRs. We deem projects active if their latest commit was made in or after 2021, and PR-welcoming if they accepted a PR from an external developer in the last month and have less than 100 open PRs. Each PR clearly explained UPGRADVISOR's goals and affiliation, provided UPGRADVISOR outputs (e.g., graphs such as the one shown in Fig. 4), and any other relevant information (e.g., dependency change log) allowing the developers to examine the updates and validate our results. In some cases, our PR prompted discussions with the developers providing us with ideas for improving UPGRADVISOR's outputs. Out of nine PRs submitted, seven were merged and two received no response. Furthermore, five of the merged PRs were for dependencies listed in Table 2, validating the results of UPGRADVISOR's dynamic tracing.

7.4 Detecting API Breakage

We noticed that in OSS projects, API breakage is discovered by dependency users in a few days/weeks. The relevant version will quickly be "yanked" from the repositories, so that the API breaking version ends up not being visible in our experiments in §7.1. As a result, none of the dependency updates considered in §7.1 caused API breakage. While this shows the advantages of OSS, for the individual entities, this discovery might have been made at the price of production failures or even data corruption, and UPGRADVISOR's goal is to detect these before they happen.

To evaluate UPGRADVISOR's ability to detect API breakage, we conducted a small controlled experiment with two applications, `django-oscar` and `label-studio`, which were examined by UPGRADVISOR in §7.1. These applications have a dependency on `Django`, which has a well-documented deprecation timeline [4] allowing us to study API breakage. We consider the recent 7-Dec-2021 release of `Django 4.0`, which contains 28 API breaking changes including arguments losing default value, removed APIs, etc. Both `django-oscar` and `label-studio` are stuck on much earlier 3.x versions of `Django`. Instead of considering an update to the next available 3.x version of `Django`, we used UPGRADVISOR to statically analyze the difference between version 4.0 and the 3.x version specified by the application. In these cases, UPGRADVISOR correctly identified all API breaking changes with no false positives or negatives, which we manually confirmed by studying UPGRADVISOR's output and comparing it to the deprecation information. This experiment also showcases UPGRADVISOR ability to direct developers to the relevant portions of their code which will break and provide context for the fix.

7.5 Tracing Overhead

We evaluated UPGRADVISOR's tracer overhead using applications from our previous experiments in §7.1 with test suites that we could set up and execute without errors. Ironically, some test suites failed to run due to broken or conflicting dependencies. We selected a subset of qualifying projects to represent the Python open-source eco-system, including ML (`Qlib` and `Flair`), data-science (`Faust`), blockchain (`Electrum` and `Vyper`), administration tools (`aws-cli`), and website-building (`Django`). `Django` allowed us to experiment with multi-process code and control the number of processes used. We ran Django's test suite using 1, 8, and 16 logical CPUs. Each project had some dependency update among the 172 possible updates considered in §7.1. Specifically, the dependency updates for `Qlib`, `Flair`, `Faust`, `Electrum`, `Vyper`, `aws-cli`, and `Django` were `hyperopt`, `gdown`, `Croniter`, `qdarkstyle`, `asttokens`, `colorama`, and `pytz`, respectively.

We compared the performance of UPGRADVISOR to several other tools, including `cProfile`, `Coverage.py`, and `JPortal4Py`. `cProfile` is a de-facto standard tool for cPython that profiles executions at the method-level. `Coverage.py` is a de-facto standard tool for cPython that tracks statement-level test coverage. Neither of them provide the same functionality of UPGRADVISOR's tracer, but provide useful performance comparisons. `JPortal4Py` is a Python-compatible implementation of a hardware tracer that traces the whole interpreter [43]. We also compared against UPGRADVISOR-SW, an implementation of UPGRADVISOR's tracer that uses software tracing in lieu of Intel PT to trace all procedures. In evaluating UPGRADVISOR, we compared two configurations, UPGRADVISOR-ALL to trace all procedures, and UPGRADVISOR-Targeted to trace only procedures marked by UPGRADVISOR's static analysis. We ran each application on each tool five times and report the average and standard deviation of the overhead measurements.

Fig. 10 shows the performance overhead measurements

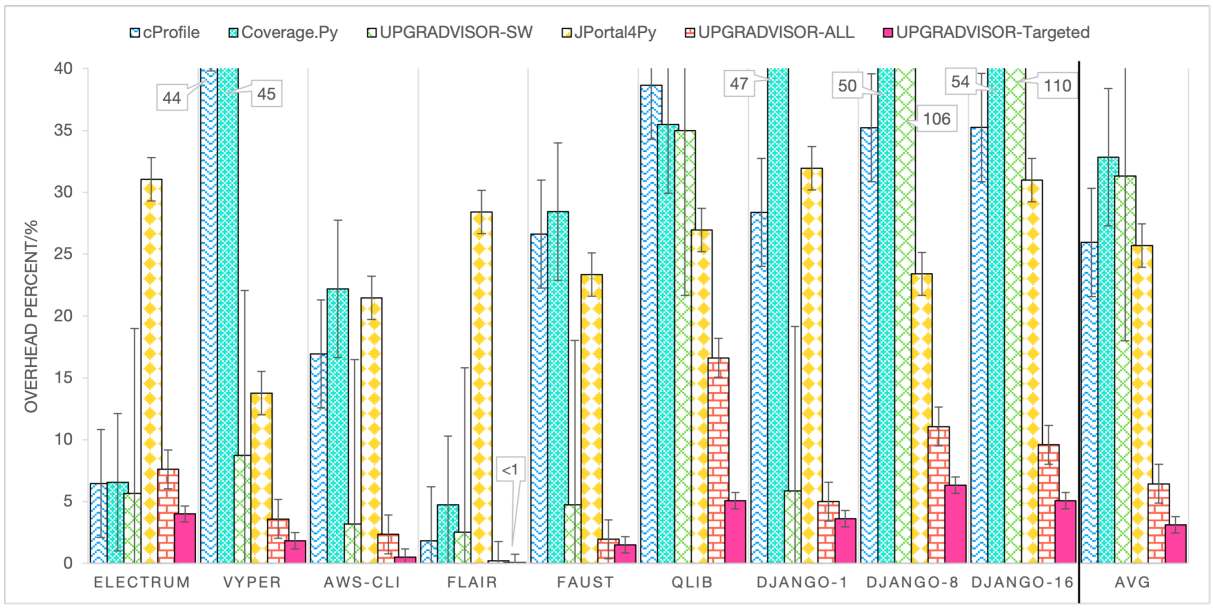


Figure 10: Comparing the performance of UPGRADVISOR's two modes, ALL and Targeted, with `cProfile`, `Coverage.py`, `JPortal4Py`, and UPGRADVISOR-SW, a software-only tracer.

normalized to native execution of the application without any tracing. UPGRADVISOR in targeted mode has the least overhead in all cases, averaging 3%, with a standard deviation of 2.15%. It is an order of magnitude faster than all other tools for some applications, except for UPGRADVISOR-ALL. Django's multiprocess test-suite measurements showcase the advantages of using hardware features for tracing, as all software-based approaches suffer from significant overhead trying to record all control operations made by the interpreter across several processes. Nevertheless hardware tracing is not a panacea as the `JPortal4Py` hardware tracer performs much worse than UPGRADVISOR-SW on most of the single process measurements. This is because `JPortal4Py` traces the whole interpreter as well, flooding the memory buffer with trace packets and causing significant disk I/O.

While tracing all methods, UPGRADVISOR-ALL manages to only incurs an average of 6.4%, over 60% worse than UPGRADVISOR-Targeted but still much better than all other tools. However, because it traces many more methods and fills up the memory buffer quickly, it suffers data loss, which can lead to misdiagnosing unsafe updates as safe. Data loss measures lost tracing events, those overwritten before they could be read from memory by the CPU and written to disk, as a percentage of all tracing events. We calculated data loss rates by comparing UPGRADVISOR-ALL versus UPGRADVISOR-SW, which also traces all methods but does not suffer the data loss of hardware tracing. UPGRADVISOR-ALL's data loss rates across the different applications rose as high as 16% for single process workloads and over 20% for Django running with 16 logical CPUs. In our experiments, we set a memory buffer size limit of 128MB per logical CPU. Increasing this limit or using faster disks/memory might help convert some data loss into overhead. In contrast, UPGRADVISOR-Targeted does not suffer from any data loss due to the reduced amount of trace records generated.

To further stress UPGRADVISOR's tracer, we used Instagram's `django-workload` [8]. This testing environment includes a Cassandra database [2], `memcached` [27] in-memory key-value instance, a Django installation and the Siege load generator [13]. We set up Django according to its recommended configuration for production systems [6] using the WSGI interface. Django depends on `pytz`, a frequently updated package dealing with time-zone related date manipulations, and supports thousands of plugins and sub-packages [5], including `django-cassandra-engine` used by `django-workload`. We measured the performance of UPGRADVISOR's tracer using `django-workload` when evaluating updates to both `pytz` and `django-cassandra-engine`. Running this workload using both UPGRADVISOR-ALL and UPGRADVISOR-Targeted, we found that UPGRADVISOR incurs an average overhead of only 7% and 3%, respectively. These results are consistent with those in Fig. 10, and indicate that our measurements of UPGRADVISOR's tracer overhead provide a good indication of its expected performance when running real-world production workloads.

8 Related Work

Dependency upgrade surveys. Other surveys also show that many projects suffer from dependency update delays [23, 38, 41]. For example, a survey of 7.3K Java projects

reports that 81.5% of projects display dependency update lag [23], and a survey of 610K JS projects in the NPM package repository between 09-11-2010 and 02-11-2017 reports a similar number of delay days as ours [41]. Our survey focuses on three modern dynamic languages and investigates historical dependency upgrade patterns.

AST differencing algorithms. AST differencing algorithms [10, 11, 16, 29] compute an edit script between two versions of an AST. GumTree [10] first finds isomorphic subtrees through a greedy top-down algorithm then executes a bottom-up algorithm to match sub-trees which share a large number of matching nodes. As discussed in §6, UPGRADVISOR uses GumTree's AST-diffing and builds upon its generated edit-script to generate a fused AST representing the dependency before and after the update.

Changeset and impact analysis. Given a set of code changes and test suite runs, change impact analysis tools generate a list of tests affected by the change and re-test them to verify if they pass after the change is adopted. Approaches can be classified based on the techniques used, the granularity of changes considered, and whether static or dynamic analysis is used [24]; only one approach explored statically studying changes at the code-snippet scope (below the method/class level) [32]. Chianti [33] introduced a change impact analysis tool for Java programs, incorporated in the Eclipse IDE. Prior techniques all rely on application test suites and do not scale to allow usage in production servers. UPGRADVISOR expands on these works, representing changes at the statement level and statically discarding them before using a dynamic tracer to validate them on production servers.

Call graph construction. PyCG [34] builds call graphs for Python code using assignment graphs. It prioritizes analysis speed and completeness and thus exhibits unsoundness in its evaluation. UPGRADVISOR prioritizes soundness, achieved by over-approximating call targets. Various approaches dynamically generate call graphs for JS code [17]. NodeProf [39] instruments the code under test and gather information in the face of code generation and other JS-born challenges. UPGRADVISOR records similar information via tracking jumps and calls online and then decoding this information offline to avoid high overhead. We plan to leverage these works to add JS support for UPGRADVISOR.

Hardware tracing. Modern CPUs provide hardware features for tracing, including Intel PT [21] and ARM embedded trace macrocell (ETM) [26, 37]). These have generally only been applicable to native programs. Our previous work, JPortal [43], showed how to enable hardware tracing for Java bytecode, but it suffers from high overhead and data loss from needing to trace the whole virtual machine. UPGRADVISOR improves on JPortal via novel coarse-grained and selective tracing mechanisms which achieve low overhead without data loss.

Statistical debugging. Statistical debugging [25, 42] reduces tracing overhead through randomized sampling and dispersing data collection among different users. UPGRADVISOR achieves low overhead through selective hardware tracing, which maintains completeness.

Multi variant execution (MVE). MVE methods [18, 28] split test suite execution at the point of change, then run the two versions (before and after upgrade) and merge them back to show compliance. MVE concepts have also been applied towards detecting exploitation attempts and test generation [22, 31]. To overcome lacking coverage in test-suites, UPGRADVISOR traces production servers focusing only on parts relevant to the dependency update.

Patch analysis in continuous integration. SubmitQueue [1] is a system for examining simultaneous application code updates. It combines a build dependency graph with a continuously trained statistical model to optimize the order of application code updates to maximize parallelism for integration tests. In contrast, UPGRADVISOR provides decision support for evaluating dependency updates using production traces.

9 Conclusions and Future Work

We have shown that many projects suffer from prolonged delays in adopting dependency updates. We have designed and built UPGRADVISOR, a system for reducing developer effort and error risk in adopting dependency updates. UPGRADVISOR features the co-design of a sound static analysis constructed to pinpoint a carefully selected target set of methods to trace and a low-overhead production-ready tracer to observe dependency usage. Using this hybrid analysis together with hardware tracing, UPGRADVISOR has analyzed 172 upgrade opportunities, determining that ~60% of them can be updated safely. For the rest, UPGRADVISOR benefits developers by reducing the manual effort of going over the changes in the dependency.

We plan to extend UPGRADVISOR to benefit more dynamic languages. Moreover, we wish to build upon UPGRADVISOR's analysis to alert about malicious updates and generate application tests for increasing dependency update coverage. We believe UPGRADVISOR's low-overhead tracing technique can become useful in other domains and intend to explore its use in debugging and fault isolation.

Acknowledgments

Landon Cox provided helpful comments on earlier drafts. Andrew Magid helped with system implementation. This work was supported in part by DARPA contract N66001-21-C-4018; ONR grants N00014-17-1-2788 and N00014-18-1-2037; NSF grants CNS-1564055, CNS-1703598, CNS-1763172, CNS-1907352, CCF-1918400, CNS-2052947, CNS-2007737, CNS-2006437, CNS-2128653, CNS-2106838, and CCF-2124080; Faculty Research Awards from Facebook, JP Morgan, DiDi, Cisco, and Accenture; and a Columbia CAIT Award. (Corresponding authors: Junfeng Yang and Zhiqiang Zuo)

References

[1] Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel, Balaji Varadarajan, Simon Soriano, Dhaval Patel, and Ali-Reza Adl-Tabatabai. Keeping master green at scale. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys '19)*, March 2019.

[2] Apache. Cassandra - open source nosql database. https://cassandra.apache.org/_/index.html. Accessed: 2022-05-24.

[3] Google Chrome. Chrome release cycle. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/process/release_cycle.md. Accessed: 2022-05-24.

[4] Django. Django deprecation timeline. https://docs.djangoproject.com/en/dev/internals/deprecation/. Accessed: 2022-05-24.

[5] Django. Django Packages is a directory of reusable apps, sites, tools, and more for your Django projects. https://djangopackages.org. Accessed: 2022-05-24.

[6] Django. How to deploy Django. https://docs.djangoproject.com/en/4.0/howto/deployment/. Accessed: 2022-05-24.

[7] Facebook Engineering. Rapid release at massive scale. https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/. Accessed: 2022-05-24.

[8] Facebook. Django workload by Instagram and Intel, v1.0 RC. https://github.com/facebookarchive/django-workload. Accessed: 2022-05-24.

[9] Facebook. Pyre: A performant type-checking for Python 3. https://pyre-check.org. Accessed: 2022-05-24.

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE '14)*, pages 313–324, September 2014.

[11] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, October 2007.

[12] Python Software Foundation. Cpython. https://github.com/python/cpython. Accessed: 2022-05-24.

[13] Jeffrey Fulmer. Siege 4.1.1 - an http load tester and benchmarking utility. https://github.com/JoeDog/siege. Accessed: 2022-05-24.

[14] Google. Atheris: A coverage-guided, native python fuzzer. https://github.com/google/atheris. Accessed: 2022-05-24.

[15] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Notices*, 32(10):108–124, October 1997.

[16] Masatomo Hashimoto and Akira Mori. Diff/ts: A tool for fine-grained structural change analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*, pages 279–288, October 2008.

[17] Zoltán Herczeg and Gábor Lóki. Evaluation and comparison of dynamic call graph generators for JavaScript. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE '19)*, pages 472–479, May 2019.

[18] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*, pages 612–621, May 2013.

[19] Intel. Intel Processor Trace virtualization enabling. https://lwn.net/Articles/737839/. Accessed: 2022-05-24.

[20] Intel. libipt: an Intel Processor Trace decoder library. https://github.com/intel/libipt. Accessed: 2020-10-31.

[21] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, chapter 35: Intel Processor Trace. June 2019.

[22] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '16)*, pages 431–442, June 2016.

[23] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, February 2018.

[24] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, December 2013.

[25] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Notices*, 38(5):141–154, May 2003.

[26] Arm Limited. *Arm® Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.5*, December 2019.

[27] memcached. memcached - a distributed memory object caching system. https://memcached.org. Accessed: 2022-05-24.

[28] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pages 907–918, May 2014.

[29] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, and Tien N. Nguyen. Operation-based, fine-grained version control model for tree-based representation. In *Proceedings of the 13th Conference on Fundamental Approaches to Software Engineering (FASE '10)*, pages 74–90, March 2010.

[30] npm. How npm works: Dependency hell. https://npm.github.io/how-npm-works-docs/theory-and-design/dependency-hell.html. Accessed: 2022-05-24.

[31] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: Testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, pages 1181–1192, May 2016.

[32] Maksym Petrenko and Václav Rajlich. Variable granularity for improving precision of impact analysis. In *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC '09)*, pages 10–19, May 2009.

[33] Xiaoxia Ren, B.G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for Java programs. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 664–665, May 2005.

[34] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. PyCG: Practical call graph generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*, pages 1646–1657, May 2021.

[35] Python steering council. Pep 318 – decorators for functions and methods. https://peps.python.org/pep-0318/. Accessed: 2022-05-24.

[36] Python steering council. Pep 484 – type hints. https://peps.python.org/pep-0484/. Accessed: 2022-05-24.

[37] Neal Stollon. *ARM ETM*, pages 213–218. Springer US, Boston, MA, October 2010.

[38] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. Technical lag of dependencies in major package managers. In *Proceedings of the 27th Asia-Pacific Software Engineering Conference (APSEC '20)*, pages 228–237, July 2020.

[39] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node.js. In *Proceedings of the 27th International Conference on Compiler Construction (CC '18)*, pages 196–206, February 2018.

[40] TIDELIFT. libraries.io - the open source discovery service. https://libraries.io. Accessed: 2022-05-24.

[41] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *Proceedings of the 17th International Conference for Software Reuse (ICSR '18)*, pages 95–110, May 2018.

[42] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pages 1105–1112, June 2006.

[43] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. JPortal: Precise and efficient control-flow tracing for JVM programs with Intel Processor Trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, pages 1080–1094, June 2021.

A Artifact Appendix

Abstract

The version of UPGRADVISOR used to perform the experiments described in the paper may be downloaded from figshare.com. The artifact contains the code for the package survey, the static analyzer, and the hardware tracer. It also contains scripts to compile the tracer, run the experiments described in the paper, and produce most of the figures. For the most up to date version of UPGRADVISOR and other resources please refer to may be accessed on Github at http://upgradvisor.github.io.

Requirements

We provide the analyzer pre-installed in a docker container. The tracer requires a bare-metal machine. It directly employs a tracing capability found in Intel 5th generation CPUs (Broadwell) and above. Installing the tracer software requires root access to the OS.

This artifact will run on a i7-10700 CPU workstation with 16GB RAM. A slower machine may result in reduced performance. We set up the docker container on the tracer machine and encourage you to do the same.

Scope

The artifact may be used to reproduce the experiments described in the paper, including Fig. 1, Fig. 4, Fig. 8, Fig. 9, Fig. 10, Table 1, and Table 2.

Contents

- AnalyzerDocker.tar.gz: A docker container for running the survey and static analysis portions of Upgradvisor.
- Cache[2].tar.gz: Cached intermediate results of Upgradvisor to serve as examples and troubleshooting aids.
- UpgradvisorArtifact-main.tar.gz: The code of the Upgradvisor analyzer and tracer.
- README.md: Instructions for setting up and running the Upgradvisor experiments.

We recommend following the README's instructions for running the survey, and static analysis, as well as for checking compatibility with, compiling, and running the hardware tracer.

Practically Correct, Just-in-Time Shell Script Parallelization

Konstantinos Kallas
University of Pennsylvania

Tammam Mustafa
MIT CSAIL

Jan Bielak
XIV Staszic High School

Dimitris Karnikis
Aarno Labs

Thurston H.Y. Dang*
MIT CSAIL

Michael Greenberg
Stevens Institute of Technology

Nikos Vasilakis
MIT CSAIL

Abstract

Recent shell-script parallelization systems enjoy mostly automated speedups by parallelizing scripts ahead-of-time. Unfortunately, such static parallelization is hampered by dynamic behavior pervasive in shell scripts—*e.g.*, variable expansion and command substitution—which often requires reasoning about the current state of the shell and filesystem.

We present a just-in-time (JIT) shell-script compiler, PASH-JIT, that intermixes evaluation and parallelization during a script's run-time execution. JIT parallelization collects run-time information about the system's state, but must not alter the behavior of the original script and must maintain minimal overhead. PASH-JIT addresses these challenges by (1) using a dynamic interposition framework, guided by a static preprocessing pass, (2) developing runtime support for transparently pausing and resuming shell execution; and (3) operating as a stateful server, communicating with the current shell by passing messages—all without requiring modifications to the system's underlying shell interpreter.

When run on a wide variety of benchmarks, including the POSIX shell test suite, PASH-JIT (1) does not break scripts, even in cases that are likely to break shells in widespread use; and (2) offers significant speedups, whenever parallelization is possible. These results show that PASH-JIT can be used as a drop-in replacement for any non-interactive shell use, providing significant speedups without any risk of breakage.

1 Introduction

The UNIX shell is an environment for composing programs from components written in a variety of programming languages. Coupled with UNIX's toolbox philosophy [41], this language agnosticism makes the shell a popular choice for succinctly expressing tasks that involve data processing, system orchestration, and other automation. Recent systems [52, 55, 63] accelerate such tasks by exploiting data parallelism: using *ahead-of-time* (AOT) analysis and transformation, these systems parse, analyze, and transform shell scripts into new scripts that execute in parallel.

Unfortunately, AOT parallelization quickly becomes intractable due to the dynamic nature of the shell: dynamic features such as variable expansion and command substitution, pervasive in shell scripts, generate and consume values at run-time while depending on and interacting with the broader environment—*i.e.*, the filesystem, the environment variables, and the shell interpreter itself. Additionally, modern shells offer several different configurations and execution modes, leading to complex behaviors described in hundreds of pages of POSIX standardese [2]. The complexity of these interactions and their side-effects lead existing parallelization tools to an unavoidable trade-off between (1) being conservative, aborting on scripts that use dynamic features, or (2) being unsound, possibly breaking scripts during parallelization. Recent systems [52, 55, 63] tend to be conservative—operating only on fully expanded shell pipelines and having a hard time even on simple uses of variables (see §2).

This paper presents PASH-JIT, a production-grade just-in-time (JIT) shell-script compiler aimed at non-interactive parallelization: PASH-JIT focuses on three practical (but conflicting) goals: (G1) run-time-informed parallelization: PASH-JIT leverages run-time information to parallelize script fragments that depend on state that is statically indeterminable; (G2) full behavioral equivalence: PASH-JIT is aware of the full set of dynamic behaviors present in POSIX shells, producing results that are indistinguishable from the sequential execution on the system's shell interpreter; (G3) loose shell coupling: PASH-JIT avoids modifications to the system's underlying shell interpreter, eschewing practical problems (*e.g.*, maintaining two Bash implementations). PASH-JIT behaves as a drop-in shell shim enhancing any non-interactive shell use, providing significant speedups without any risk of breakage.

PASH-JIT's key insight is to parallelize scripts just-in-time: by intermixing evaluation and parallelization during a script's execution, PASH-JIT collects and uses the latest possible run-time information about the state of an expression's vari-

*The author is now at Google but the work was done while he was at MIT.

ables, the shell, and the filesystem. PASH-JIT parallelizes script fragments when it is safe to do so, resolving indeterminacies in the broader environment on the fly. Unfortunately, low-overhead run-time-informed parallelization (G1) is particularly challenging to implement in view of full behavioral equivalence (G2) and loose shell coupling (G3). PASH-JIT addresses this conundrum using: (1) a dynamic interposition framework, guided by an instrumentation preprocessing pass; (2) support for reentrance, transparently pausing and resuming the execution of the underlying shell interpreter at run-time; and (3) a stateful, long-lived compilation server that communicates with the current shell by exchanging messages. A 9K-LOC implementation and several run-time optimizations—*e.g.*, dynamic independence discovery, commutative-aware parallelization—complete the picture.

We apply PASH-JIT to a variety of benchmarks, ranging from scripts collected from the wild to the POSIX test suite. PASH-JIT behaves identically to Bash 4.4.20(1) on 406 out of 408 applicable POSIX tests; matching Bash is a significant achievement even for a non-parallelizing shell—shells in widespread use differ on much larger subsets of tests. PASH-JIT offers speedups up to 33.7× over Bash on a 64-core machine (improving the state of the art [63] by 2× on average), notably parallelizing scripts that prior work failed to parallelize due to dynamic behaviors.

The paper begins by exemplifying dynamic shell features and the application of PASH-JIT's techniques (§2). Sections 3–6 describe PASH-JIT's main contributions:

- *A dynamic interposition framework for the shell*: A just-in-time analysis and optimization subsystem enables safe and effective parallelization during the execution of a script, dealing with the challenges of dynamic shell-script behavior. A first pass determines where to insert calls to a parallelizing optimizer in a given input script (§3), which is then invoked on-the-fly while the script is executing (§4).
- *A stateful, parallelizing compilation server*: PASH-JIT queries a long-lived parallelization server at run-time to compile script fragments. This model improves run-time efficiency by avoiding startup costs on every JIT invocation, and enables additional run-time optimizations for (1) executing independent regions in parallel, and (2) pipelining compilation and execution. The core of the server has been modelled and formally verified using SPIN [29] (§5).
- *Commutativity-aware optimization*: Additional compilation optimizations target commands that are commutative with respect to their input, along with parallelizing transformations and run-time primitives that improve the run-time performance of scripts that contain such commands (§6).

The paper then presents PASH-JIT's evaluation (§7) and related work (§8), before concluding (§9). PASH-JIT is MIT-licensed open-source software supported by the Linux Foundation at https://github.com/binpash/.

2 Example & Overview

Below is a shell program that downloads a compressed archive of text files (books from Project Gutenberg), extracts them in a directory, and then performs an analysis to find the frequencies of all words of a specific form.

```
IN=${IN:-$TOP/pg}
mkdir "$IN"
cd "$IN"
echo "Download will take some time, be patient..."
wget "$SOURCE/data/pg.tar.xz"
if [ $? -ne 0 ]; then
  echo "Download failed!"
  exit 1
fi
cat pg.tar.xz | tar -xJ

cd "$TOP"
OUT=${OUT:-$TOP/output}
mkdir -p "$OUT"
for input in $(ls "$IN"); do
  cat "$IN/$input" | tr -sc '[A-Z][a-z]' '[\012*]' |
    grep '^....$' | sort | uniq -c > "$OUT/$input.out"
done
```

The program makes pervasive use of the shell's dynamic features. For example, it uses environment variables such as `$TOP`, variable expansion like `${OUT:-$TOP/output}` to assign default values, command substitution `$(...)` as part of the loop condition, and state reflection on the file system by running `ls` on `$IN` (itself resolved dynamically).

None of the values of these variables can be known ahead of time just by analyzing the program's source code. They become known only at run-time, when the shell interpreter reaches these points in the program's execution. A sound AOT compiler such as PASH-AOT [63] or POSH [52] would fail to parallelize—foregoing all the performance benefits of data-parallel execution spread across many files in `$IN`.

PASH-JIT instead takes a JIT approach that interjects parallelization opportunities during and throughout the script's execution (Fig. 1).

Dynamic interposition (§3): PASH-JIT first uses a preprocessing step to instrument all potentially optimizable program regions with calls to the JIT engine. PASH-JIT chooses regions to maximize the potential benefits of parallelizing them: intuitively, commands and pipelines can yield significant benefits, whereas word expansion, control flow, and variable assignments are operations that do not perform heavy computation and can therefore be left as they are. PASH-JIT's preprocesor and compiler both make extensive use of parsing/unparsing of shell source code, implemented as a new parsing library. After PASH-JIT has inserted calls to the JIT engine, it invokes the user's shell interpreter to execute this transformed script. During this execution, the JIT engine calls the parallelizing compiler at run-time—right before the execution of each fragment, when the state of the shell and the file system have already been resolved. The transformed program

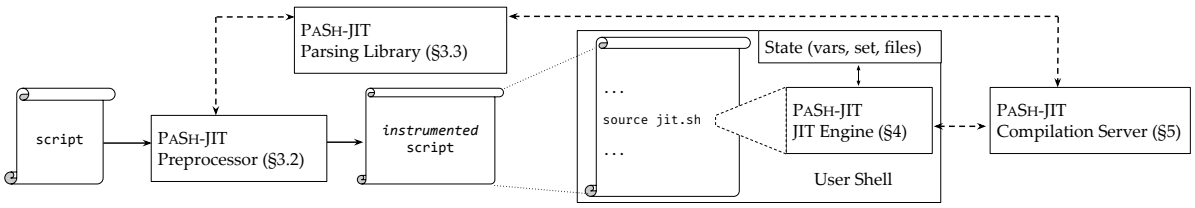


Fig. 1: PASH-JIT overview. PASH-JIT instruments scripts with calls to the JIT engine, which passes program fragments to the compilation server at run-time.

maps original commands to *regions*—for example, `region8` corresponds to the `cd` call and `region10` corresponds to the pipeline in the **`for`** loop.

```
source jit.sh "$region8"                    # cd $TOP
OUT=${OUT:-$TOP/output}
source jit.sh "$region9"              # mkdir -p "$OUT"
for input in $(ls "$IN"); do
  source jit.sh "$region10" # cat "$IN/$input" | ...
done
```

The command `source jit.sh "$regionN"` invokes the JIT engine passing as argument the corresponding fragment. The `source` built-in retains the same shell environment, reflecting any effects directly into the current environment.

JIT engine (§4): Internally, the JIT engine first saves the state of the shell at that point in the script's execution to isolate it from compilation—protecting the shell from the JIT engine and protecting the JIT engine from obscure shell configurations. PASH-JIT then invokes the compiler to attempt to parallelize the fragment. If the compiler succeeds, PASH-JIT runs the resulting parallel fragment; if not, it runs the original, unmodified region. In both cases, PASH-JIT will first restore the state of the shell before executing the fragment. Whether the compiler succeeds or not depends on the properties of the fragment's code—*e.g.*, PASH-JIT will reject `region8` due to the side-effectful `cd` command, but will accept `region10` compiling `grep` and `sort` into the parallel fragment below:

```
c_split /tmp/fifo8 /tmp/fifo9 /tmp/fifo10 &
c_wrap 'grep "^....$"' </tmp/fifo9 >/tmp/fifo11 &
c_wrap 'grep "^....$"' </tmp/fifo10 >/tmp/fifo12 &
c_strip </tmp/fifo11 >/tmp/fifo13 &
c_strip </tmp/fifo12 >/tmp/fifo14 &
sort </tmp/fifo13 >/tmp/fifo15 &
sort </tmp/fifo14 >/tmp/fifo16 &
eager.sh </tmp/fifo15 >/tmp/fifo17 &
eager.sh </tmp/fifo16 >/tmp/fifo18 &
sort -m /tmp/fifo17 /tmp/fifo18 >/tmp/fifo19 &
```

The resulting compiled fragment executes in a data-parallel fashion: data is split by PASH-JIT primitives, then fed to multiple instances of `grep` and `sort` runnning in parallel, and finally merged at the end of the parallel execution.

Dependency untangling (§5): While the JIT engine operates as if invoked on every region, PASH-JIT is engineered to spawn a long-running stateful compilation server just once, feeding it compilation requests until the execution of the script completes. This design has two benefits: (1) it reduces run-time overhead by avoiding reinitializing the compiler for each compilation request; and (2) it allows maintaining and querying past compilation results when compiling a new fragment. The latter allows PASH-JIT to untangle dependencies *across* regions, finding and exploiting opportunities for cross-region parallel execution. For example, the server's first invocation on `region10` (the body of the loop) determines that all prior successfully compiled regions have finished executing. PASH-JIT can thus simply run the loop in the background and continue with the second iteration in a task-parallel fashion, without waiting for the first iteration to complete executing. During the second invocation on `region10`, PASH-JIT will use the dependency state to determine that while the previously compiled fragment is still running, the input and output files of the two regions are completely independent and can thus be executed in parallel: our loop is now pipelined! PASH-JIT goes beyond intra-region data parallelism: the JIT enables inter-region task parallelism by resolving dependencies and confirming they are independent.

Commutativity analysis & compilation (§6): The first goal when compiling fragments such as `region10` is to identify command sequencies that are parallelizable using a divide-and-conquer strategy. Due to the shell's order-aware nature [28], naive divide-and-conquer would need to (1) read the entire input before splitting it, to determine the exact size of each batch, leading to stalled pipeline parallelism; and (2) wait until all of its predecessors have consumed their batch, storing data after split on disk, to ensure that all parallel nodes will not wait for their input.

While these overheads are unavoidable in the general case, and are indeed incurred by prior systems [55, 63], they can fortunately be alleviated for subsets of parallelizable commands. Two such subsets include (1) stateless commands such as `grep -c '^....$'` that operate in a line-oriented fashion, meaning that data-parallel copies of these commands can combine their partial output using a reordering operation, and (2) commutative commands such as `sort -u` that produce equivalent output regardless of the order of the input lines. PASH-JIT leverages this insight to achieve more effective parallelization by splitting into streaming micro-batches (using `c_split`) in a round-robin fashion—avoiding the overheads of reading all the input before splitting and of unnecessary storage to disk. It also wraps stateless commands to strip and re-add the microbatch headers (using `c_wrap`) and removes these headers completely before commutative commands (using `c_strip`).

Zooming back out: Fundamentally, PASH-JIT is neither a shell nor requires modifications to a user's shell. Rather, it is an interposition shim located between a user and their shell, deciding whether to optimize parts of the user script on the fly, using information about the execution state of the shell interpreter. PASH-JIT combines several techniques that allow harnessing speedups not attainable by ahead-of-time parallelization on both dataflow-only scripts and larger scripts with dynamic components and complex control flow; all of this, without modifying the behavior of the original script.

3 Interfacing With the Shell

PASH-JIT works by interposing on the shell, effectively rewriting invocations to external commands. Challenges arise due to the shell's complex semantics and its intricate internal state, both of which complicate side-effect-free interposition. The shell uses a string-based, bi-modal semantics: commands undergo *expansion*, a string rewriting phase where variables, tildes, and globs are processed before the commands undergo *evaluation*. Both modes have complex semantics heavily involved with the shell's state [24]; any rewriting must be careful to leave the shell's state unaltered.

3.1 Dynamic Interposition

To understand PASH-JIT's interposition, we must first understand the simpler structure of ahead-of-time (AOT) parallelization. While preserving a script's original behavior, AOT parallelization rewrites calls to external commands to exploit parallelism. External commands consume substantially more time and resources than shell language features (like expansion or loops) during the execution of typical shell scripts.

AOT parallelization centers around the identification of *parallelizable regions*—script fragments that may be safely parallelized to yield performance gains. Semantically, parallelizable regions only contain a set of command invocations that satisfy the following conditions: (1) they have no file dependencies (*interference-free*), *i.e.*, all commands can execute concurrently without affecting each other, (2) they communicate with each other using explicit UNIX channels (fifos/pipes); (3) they are *pure*, only affecting the environment by reading and writing to files, *i.e.*, they do not modify environment variables; and, (4) they are fully expanded. An AOT compiler parses and transforms these regions to an intermediate representation such as directed-acyclic [52] or dataflow [63] graphs, abstracted as functions that take a set of input files and produce a set of output files [28]. It then applies transformations on these graphs to perform the original computation in parallel.

PASH-JIT works similarly, but applies these steps at a much finer granularity and in a dynamic, online fashion. PASH-JIT's dynamic interposition mechanism pauses execution right before each parallelizable region, compiling it to an efficient and equivalent parallel script fragment, and executing that instead. Working dynamically means PASH-JIT has up-to-date information and can achieve increased parallelism.

3.2 Preprocessor

Dynamic script interposition without any shell-interpreter modifications is hard. To achieve this, PASH-JIT opts for a light-weight script instrumentation pre-processing step: it marks *possible* parallelizable regions with code that dynamically determines whether or not to invoke the compiler.

The intuition behind PASH-JIT's preprocessor is that a syntactic analysis of a shell script is enough to suggest potential parallelizable regions. This analysis is imprecise: there is no way to determine whether a command invocation will be pure ahead of time. Its goal however, is not to find parallelizable regions exactly, but rather to find potential compilation sites—PASH-JIT sorts out the details at run-time, using up-to-date information about the system's state.

There is a trade-off when choosing the right size for these regions: the larger the region, the more opportunities exist for analysis and optimization but the less likely it is for the entire region to be parallelizable. PASH-JIT targets a middle-ground: maximal syntactic schedule-free regions—*i.e.*, command sequences composed using shell primitives that do not impose scheduling restrictions. By focusing on maximal schedule-free regions, PASH-JIT minimizes the number of compiler invocations and maximizes the cross-command parallelization opportunities for the compiler. Note that schedule-free regions underapproximate interference-free regions (§3.1), *e.g.*, two commands composed in sequence `;` that write to different files do not interfere but are not syntactically schedule-free.

The preprocessor finds these maximal regions by searching the AST bottom-up, combining schedule-free subtrees when they are composed using constructs that do not introduce scheduling constraints (*e.g.*, `&`, `|`). When a region cannot outgrow a certain subtree, it is replaced with a call to the JIT engine. If successfully compiled, a region is transformed to a dataflow graph—a convenient and well-studied computation model amenable to transformation-based optimizations [28]. The instrumented AST resulting from the compilation is finally translated (unparsed) back to shell code and sent over to the underlying shell for execution.

3.3 Parsing Library

Parsing and unparsing are key operations in PASH-JIT and must address several challenges.

PASH-JIT parses lines of shell script as they come in, and unparses lines in order to execute them in the user's shell; it also uses parsing and unparsing during compilation, when the compilation server emits an optimized string or passes

strings to the shell for expansion. PASH-JIT initially used `libdash`—an OCaml library built using the dash parser and part of Smoosh [23, 24]—that caused two main issues. First, `libdash`'s unparsing introduced several bugs, as at the time it was used by the libdash project primiarly for testing and diagnostics—had much of its was functionality untested. Second, `libdash` parsing introduced significant run-time overhead due to (1) the cost of forking and executing the OCaml binary, (2) overheads due to serialization and deserialization during communication, and (3) suboptimal implementation. Run-time overheads were a significant concern due to PASH-JIT's online JIT parallelization, which intermixes calls to the compiler during the program's execution—bringing parsing and unparsing into the critical path of program execution.

To address these issues, PASH-JIT reimplements its own version of `libdash` in Python called Pylibdash. The Pylibdash implementation develops Python bindings for the dash parser and completely reimplements unparsing—adding 0.9k LOC of Python over libdash, structured as a separate library usable by other projects. The Pylibdash implementation contains several optimizations such as caching, inlining, and careful array appending to avoid some accidentally quadratic costs in the original implementation. As a side benefit, using a custom implementation reduces the number of dependencies required by PASH-JIT's installation.

4 The JIT Engine

The PASH-JIT preprocessor identifies possible parallelizable regions and instruments the shell script to dynamically determine whether they can be optimized by invoking the JIT engine. The JIT engine faces two key challenges: it must not change the original script behavior, and it must run with low overhead as it is invoked multiple times per script.

The JIT engine is a reflective shell script: by inspecting the state of the shell and that of the broader system, it can transparently work with the compiler to determine whether or not to parallelize a script (Fig. 2). When running scripts with PASH-JIT, it is helpful to think of the shell as having two modes: (1) conventional shell mode, where scripts execute in the original shell context, and (2) PASH-JIT mode, where the runtime reflects on shell state and invokes a compiler to determine whether to execute the original or an optimized version of the target region. To switch from shell mode to PASH-JIT mode, the JIT engine must carefully save the state of the user's shell; to switch back, it must carefully put things back just the way they were. A shell's state is quite complex: beyond saving and restoring variables, the runtime must account for various shell flags along with other internal shell state (*e.g.*, the previous exit status, working directory).

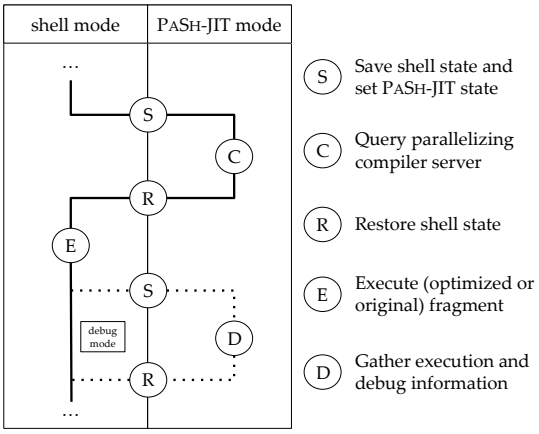


Fig. 2: Overview of JIT engine stages.

4.1 JIT Stages

When running normally, the JIT engine transitions into and out of PASH-JIT mode once per possible parallelizable region (Fig. 2): the JIT engine saves the shell state and switches into PASH-JIT mode (S); then it tries to compile the current fragment (C); whether successful or not, the JIT engine restores the state and switches back to shell mode (R); and, finally, either the original fragment or the optimized parallel version is executed (E). With debugging enabled, the JIT engine switches back into PASH-JIT mode (S) to collect debugging information (D), restoring again afterwards (R).

Saving (S): When entering a possible parallelizable region, the first step is to save the shell state—recording the previous command's exit status, the values of environment variables, and the configuration of the shell—essentially, a continuation that can later be restored to execute the target fragment. Once the state is saved, PASH-JIT mode reconfigures the user's shell to avoid changing script behavior. For example, if the user's shell has the `-e` "exit on error" flag set, the shell should exit immediately when a command (or a pipeline) returns a non-zero exit status, unless that command is in a checked position (*e.g.*, after `!`, or in the condition of an **`if`** or **`while`**) [2]. However, failing commands should not stop the JIT itself, so `-e` is unset (and will be restored later in (R)).

Compilation (C): With the state saved and shell reconfigured, PASH-JIT tries to compile the script fragment: the JIT engine queries the compilation server (§5) with the script fragment (already parsed during preprocessing) along with the saved shell state, so that the compilation server can try to expand all of the words in the fragment. The server responds to indicate whether it managed to optimize the fragment.

Restoring (R): Whether or not compilation was successful, the JIT engine exits PASH-JIT mode, restoring the continuation saved earlier (S) to prepare to execute the fragment. One particular challenge in this mode is to restore state while

accommodating different shell modes. Suppose PASH-JIT is in `-e` mode, trying to run some possible parallelizable region, and the command *before* this region exited with status 47 in a checked position, *i.e.*, without forcing the shell to exit. The JIT engine saves the exit status so as to not overwrite it. The fragment may depend on the exit status, so PASH-JIT needs to restore it before running the fragment. But it must be careful—simply running `(exit 47)` would force the shell to exit. Thus PASH-JIT runs the subshell in a checked position:

```
if (exit "$pash_previous_exit_code"); then
    source "$fragment"; ...
else
    source "$fragment"; ...
fi
```

This odd code ensures that the fragment (in identical branches) has access to the previous exit status (in the checked, conditional position of the **`if`**) without exiting when `-e` is set.

Execution (E): Back in shell mode, the JIT engine executes the fragment. If the compiler was successful, then the JIT engine selects the optimized script fragment. If the compiler failed, the JIT engine falls back to the original fragment. Either way, control flows back to the original shell.

Debug mode (S) (D) (R): When PASH-JIT is in debugging mode, the JIT engine will re-enter PASH-JIT mode after execution (E) in order to log information about the script, such as execution time and exit status. Standard execution skips this extra save/restore cycle.

5 Parallelizing Compilation Server

For each possible parallelizable region, the JIT engine queries the compiler: can this region actually be optimized? To answer this question, PASH-JIT builds on ideas from the PASH-AOT [63] dataflow compiler (§5.1). As ever, it focuses on preserving behavior and minimizing overhead.

To preserve correct behavior in the face of the shell's dynamism, PASH-JIT expands each script region prior to compilation (§5.2). To minimize overhead due to fixed startup costs—*e.g.*, initialization, dependency loading, logging setup, and output file arrangement—PASH-JIT packages the new compiler as a stateful compilation server communicating via UNIX domain sockets.[1]

The compilation server is also augmented to support a larger set of optimization opportunities, by storing and using information from one compilation to help another. PASH-JIT's long-lived compilation server achieves these additional optimizations by allowing parallelizable regions that work on independent inputs and outputs to be run in parallel (§5.3) and by learning to improve its parallelism configuration from past compilations (§5.4).

[1] We experimented with both socket and FIFO-based communication, but we saw no significant performance differences.

5.1 Command Annotations

PASH-JIT uses the command annotation and specification framework introduced by PASH-AOT [28, 63], extended to also indicate whether a command invocation is commutative (§6.1). This framework provides information about a command invocation's parallelizability class, inputs, and outputs. A command annotation can be used to extract high-level information about a specific command invocation, *i.e.*, a precise instantiation of its flags, options, and arguments. For example, annotations determine whether a given command invocation is pure and what its inputs and outputs are.

PASH-JIT uses this annotation framework to extract information for commands that are not shell builtins—that is, commands like `sort` and `grep`. Annotations enable analyses and transformations over command invocations by lifting them to pure dataflow nodes in a dataflow intermediate representation (IR) [28]. For example, `grep -f dict.txt src.txt > out.txt` is a dataflow node with two input files (`dict.txt` and `src.txt`) and one output file (`out.txt`), which are all extracted from the annotation of the `grep` command. Annotations also describe parallelization opportunities, *e.g.*, `grep "pattern" src.txt` processes each line of `src.txt` independently, and so it can be parallelized.

5.2 Early, Pure Expansion

PASH-AOT can only attempt to compile script fragments where all words are completely expanded. Running dynamically, PASH-JIT goes beyond PASH-AOT by expanding words according to the current state of the system (shell, file system, *etc.*).

One way to achieve expansion would be for PASH-JIT to maintain a "mirror" Bash process when initializing, which it could then query with any word to expand using `echo`. Every time PASH-JIT would query the compilation server with a fragment, it would also provide the latest state of the shell, which would in turn be passed to the mirror process to ensure it reflects the latest state. This expansion method would be correct, as it would leverage the underlying shell. It would, however, be expensive, since each fragment contains many unexpanded words and each unexpanded word would have to be expanded using its own `echo` command—leading to unnecessary run-time costs.

PASH-JIT avoids the overhead of a mirror shell by performing its own expansion, relying on the optimistic nature of the JIT engine (§4): if most common forms can be expanded in the compiler itself, the compiler will succeed often without incurring interprocess communication overheads; if expansion fails, PASH-JIT will just run the original fragment. Armed with this insight, PASH-JIT implements a subset of expansion in the compilation server itself. PASH-JIT's custom expansion is *purely* functional, in that it does not affect shell state by setting variables or running command substi-

tutions. The expansion routine is implemented in less than 300 LOC of Python, and reduces the compilation overhead significantly (§7). Expansion takes the host shell's configuration and expands common, safe expansions in as many positions as possible—in simple commands, pipelines, and other parallelizable regions.

PASH-JIT's expansion routine implements most parameter formats, plain tildes, and appropriate quoting. Currently, it does not cover impure expansion (*e.g.*, parameter formats that have side-effects like `${x=foo}`, which will set `x` to `foo` if `x` is unset), since impurity violates the parallelizable region requirements. It also does not implement a few expansion cases—*e.g.*, arithmetic expansions of the form `$((x + 1))`—that were not seen in the corpus of parallelizable scripts used to evaluate PASH-JIT (§7). Adding support for unimplemented forms would require engineering effort, but not a fundamental change to PASH-JIT's expansion. If the expansion encounters a term it cannot expand—because it is unimplemented or because it would be impure—the compilation process aborts and PASH-JIT runs the original fragment.

5.3 Dependency Untangling

PASH-JIT's compilation server makes it easy to detect when parallelizable regions are independent—including, for example, independent program fragments that are sequentially composed with `;` or different iterations of a `for` loop. A key insight here is the semantics of PASH-JIT's successful compilation: if the PASH-JIT compiler succeeds on a given region, that region's original script fragment must only affect its input and output streams (files). That is, successful fragment compilation means that the fragment is *pure*, reading from and writing to a well-defined set of streams without modifying any other global system state such as non-temporary streams or environment variables.

The PASH-JIT compiler thus tracks each parallelizable region in terms of its read and write sets, which suffice to detect read-write and write-write dependencies between fragments. If two fragments (a) compile successfully and (b) have no dependencies, they can be executed in parallel. This optimization improves performance not only because of the parallel speedup, but also because it overlaps (*i.e.*, pipelines) compilation and execution, reducing net run-time overhead.

To discover independent fragments, the compilation server (Fig. 3) and JIT engine (Fig. 4) are extended to communicate about successfully compiled fragments. Coordinating using `exit` requests, the compilation server maintains a map of running fragments. When it receives a compilation request that succeeds, the server waits for all prior fragments with dependencies to finish executing; only then does it send the compiled fragment to the JIT engine for execution in the background. While the compiled fragment executes in the background, the JIT engine can exit PASH-JIT mode, and execution proceeds with the rest of the input script. When

```
# State contains a map from ids to
#   inputs and outputs.
while True:
  req = receive_request()
  if reached_script_end(req):
    wait_all()
    exit()
  else if is_exit_request(req):
    state.remove_id(req.id)
  else if is_compile_request(req):
    compile_res = compile(region)
    if not compile_res.success:
      wait_all()
      respond(compile_res)
    else if compile_res.success:
      # Wait until all ids with dependencies
      #   finish executing.
      wait_for_dependencies(compile_res.inputs,
                            compile_res.outputs)
      request_id = fresh_id()
      state.add_request(request_id, compile_res)
      respond(compile_res, request_id)
```

Fig. 3: Compilation server algorithm (pseudocode) extended for dependency untangling (*Cf.*§5.3).

```
# Blocking query
res = query_server(compile_request(region))

if res.success:
  # Run the compiled code in parallel
  fork({
    run(compiled)
    send_exit(res.id)
  })
else:
  run(original)
...
```

Fig. 4: JIT engine algorithm (pseudocode) extended for dependency untangling (*Cf.*§5.3).

execution reaches another fragment and the JIT engine returns to PASH-JIT mode, the JIT engine will block again until the compilation server responds. Even if the compilation server encounters a fragment that fails to compile, the server blocks on dependencies: the uncompilable fragment might have arbitrary side-effects.

To ensure that our algorithm is correct, we modeled it using the SPIN Model Checker [29] and we verified (i) that it does not lead to deadlocks, (ii) that no failed compiled region is running simultaneously with any other region, and (iii) that two regions with dependencies never run at the same time.

5.4 Profile-driven Compiler Configuration

The long-lived PASH-JIT compilation server can additionally use dynamic information to improve compilation. One particularly effective optimization is to dynamically determine maximum parallelism degree. As scripts might already fea-

ture task-based parallelism, spawning too many data-parallel processes can overload the system—leading to higher overheads that cut into the speedup or even result in a slowdown. These slowdowns tend to occur when there are many computationally light commands with small inputs, *i.e.*, when the overhead of managing parallelism is higher relative to the actual work to be done. The PASH-JIT compiler can reflect on prior fragments to determine an appropriate parallelism degree.

The compilation server is often queried to compile the *same* fragment many times—*e.g.*, in each iteration of a loop. At run-time, the compiler collects and maintains execution-time information. As program fragments are recompiled, PASH-JIT tries progressively narrower parallelization degrees in an attempt to minimize overall execution time.

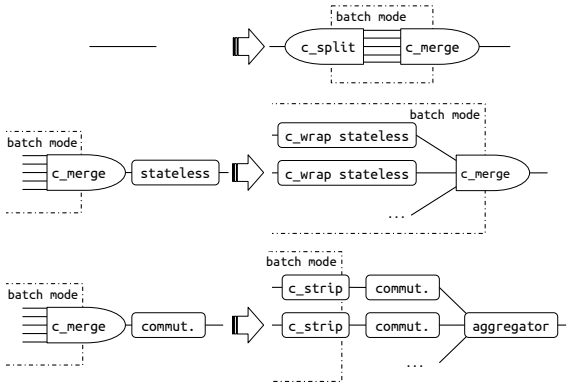


Fig. 5: Overview of commutativity-aware transformations.

6 Commutativity Awareness

Commutative commands can improve parallelization gains by allowing PASH-JIT to split and process data-parallel partial inputs in small and order-independent batches. Splitting input into many small batches improves expected CPU utilization and allows for additional pipeline parallelism. CPU utilization is improved due to an increase in partial input batches: the more work items, the more uniform the work each parallel copy does. Additional pipeline parallelism is achieved by overlapping input splitting and processing: rather than reading the entire input before deciding how to split it into batches, input can be split via small incremental steps that are immediately handed off to data-parallel commands for processing.

The PASH-JIT compiler uses these insights to produce more efficient parallel implementations of scripts that contain commutative commands. It introduces a few auxiliary nodes in its intermediate representation (IR) that orchestrate parallel execution for stateless and commutative commands, and compiler transformations that insert these nodes in a dataflow graph. It also provides efficient primitives implementing these nodes when instantiating in the parallel target script.

6.1 Compilation: Dataflow Model

The PASH-JIT compiler operates on a dataflow IR that builds on PASH-AOT, where commands correspond to nodes and communication channels correspond to edges between nodes. To enable commutativity-aware transformations, PASH-JIT extends PASH-AOT's annotation framework (§5.1) to indicate whether a command invocation is commutative (in addition to its parallelizability characteristics).

Command nodes: PASH-JIT introduces the following four dataflow nodes, which correspond to PASH-JIT-provided binary commands available in the `PATH`: `c_split`, `c_wrap`, `c_strip`, and `c_merge`. The `c_split` node takes a single input stream and N output streams. It splits its input into small batches, prepends a header on each batch identifying its sequence number, and then forwards it to one of the N outputs depending on a load-balancing strategy. Currently, PASH-JIT implements a round-robin strategy. The `c_merge` node performs the inverse operation: it merges N input streams into one and removes any headers. The `c_wrap` command is used to wrap stateless commands. It removes the header, forwards the input to the command, and then adds the header back to the command output. Finally, `c_strip` is a single-input-single-output header-removal node that often precedes commutative commands.

Transformations: To expose commutativity-aware parallelism, PASH-JIT transforms the dataflow graph; see §2 for an example. The transformations are visualized in Figure 5. The first transformation introduces a pair of `c_split` and `c_merge` before any commutative (*e.g.*, `sort`) or stateless (*e.g.*, `grep`) command. Another transformation then tries to eliminate unnecessary splits and merges, delaying `c_merge` as late as possible (*i.e.*, enclosing the biggest possible part of the graph). If a stateless command follows a `c_merge`, the command is wrapped with `c_wrap` and the `c_merge` is commuted after it. If a commutative command follows a `c_merge`, the command is parallelized and `c_merge` is transformed to a set of `c_strip` commands. Finally, if a `c_split` follows a `c_merge`, then the two are fused together to the identity function, connecting the inputs of `c_merge` with the outputs of `c_split`.

An important execution invariant is that `c_split` and `c_merge` (or `c_strip`) satisfy the requirements of well-formed parentheses, *i.e.*, a `c_split` must always be followed by a `c_merge` or a set of `c_strip` commands. PASH-JIT's dataflow graphs are essentially bimodal, since subgraphs that are between a `c_split` and a `c_merge` will execute with batches, requiring all commands in them to be wrapped with `c_wrap`, while the rest of the dataflow graph executes like the original.

Tab. 1: Benchmark summary. Summary of all the benchmarks used to evaluate PASH-JIT and their characteristics.

	Benchmark Set	Short Label	Sections	Scripts	LOC	Input	Source
1	POSIX Test Suite	PosixTests	§7.1	7	29k	—	[26]
2	Common & Classic One-liners	Classics	§7.1–7.3	10	123	14G	[6,7,33,41,59]
3	Bell Labs Unix50	Unix50	§7.1–7.3	36	142	21G	[8,37]
4	COVID-19 Transit Analytics	COVID-mts	§7.1–7.3	4	79	3.4G	[62]
5	Natural-Language Processing	NLP	§7.1–7.3	21	306	1060 books	[15]
6	NOAA Weather Analysis	AvgTemp	§7.1–7.3	1	31	36.2G	[65]
7	Wikipedia Web Indexing	WebIndex	§7.1–7.3	1	116	1000 files	[63]
8	Video/Audio Processing	MediaConv	§7.1–7.3	2	35	2.2+2.2G	[52,56]
9	Program Inference	ProgInf	§7.1–7.3	1	18	2330 libraries	[64]
10	Traffic/PCAP Log Analysis	LogAnalysis	§7.1–7.3	2	63	10–20G	[52,56]
11	Genomics Computation	Genomics	§7.1–7.3	1	34	100G	[11,51]
12	AUR Package Compilation	AurPkg	§7.1–7.3	1	27	150 packages	[13]
13	Encryption/Compression	FileEnc	§7.1–7.3	2	44	20G	[43]
14	Microbenchmarks	MicroBench	§7.3	1	6	—	custom (ours)

6.2 Runtime: Commutativity Implementation

The runtime splits the source in small batches (that contain complete lines) in a round-robin fashion.

Protocol: To reconstruct the order of different outputs while merging, PASH-JIT needs to keep track of ordering as input batches are sent to different command copies for processing and, more generally, as input-output batches flow throughout the parallelized script. To achieve this, PASH-JIT wraps all input batches with a header that contains the three following fields: `block_id`, for ordering blocks; `block_size`, the size of the block in bytes; and `is_last`, a boolean value true only for the last block with a given `block_id`.

Utilization and deadlocks: PASH-JIT must avoid deadlocks during write operations between the wrapper commands and the commands they wrap—*i.e.*, the two should never be blocked trying to write at the same time. Additionally, the wrappers must maximize utilization of the command they wrap, *i.e.*, they should never wait on input unnecessarily. To avoid deadlocks, PASH-JIT wrappers use non-blocking read and write; and to increase utilization and reduce waiting time, they write in small chunks of 32KB.

Handling inputs with long lines: An input may contain lines that are longer than the `c_split` block size. Such an event leads to non-uniform block sizes and high memory consumption, because each block must be read and sized completely before splitting and adding to the header. PASH-JIT addresses this issue by introducing the `is_last` header field in `c_split`: if a block exceeds the specified size (due to containing large lines) the block is split into multiple blocks; all blocks share the same `block_id` but only the last sets `is_last` to true. Sub-blocks with the same `block_id` are sent downstream in-order, and therefore downstream commands can use the `is_last` information to correctly reconstruct the output and know when a block ends. Block splitting reduces memory requirements and improves performance, as it allows for higher utilization regardless of the frequency of newlines. And blocks maintain a constant size throughout the flow, despite the presence of commands with high output-to-input ratio such as `curl`.

Handling small inputs: Inputs that are smaller than `c_split`'s block size lead to a single block and thus sequential execution. PASH-JIT's `c_split` addresses this issue by first attempting to read an input size s equal to `downstream_count * block_size` bytes before forwarding any blocks. If the total input is larger than s, this buffering ensures that all parallel instances will get at least one block; if the total input is smaller than s, then the input read is re-split into blocks fairly and forwarded downstream. The size s is configurable and defaults to 1MB, which we empirically determined avoids both high overhead and low utilization.

7 Evaluation

The PASH-JIT implementation comprises 6784 lines of Python (preprocessor, compilation server, expansion, compiler, and parser), 1011 lines of shell code (JIT engine and various utilities), and 1174 lines of C (commutativity primitives, and other runtime components). All line counts are of semantically meaningful lines only.

To evaluate PASH-JIT, we use three experiments on benchmarks (Tab. 1). The first experiment focuses on PASH-JIT's compatibility and uses the entire POSIX test suite as well as additional scripts (§7.1). The second experiment focuses on the performance gains achieved by PASH-JIT's parallelization, evaluated using a variety of benchmarks and workloads (§7.2). The last experiment zooms into PASH-JIT-internal overheads and associated optimizations (§7.3).

Hardware & software setup: PASH-JIT was run on 64 physical × 2.1GHz Intel Xeon E5-2683 cores with 512GB of RAM, Debian 4.9.144-3.1, GNU Coreutils 8.30-3, GNU Bash 4.4.20(1), and Python 3.7.3. There is no special configuration in hardware or software. We use Dash v.0.5.8-2.10 and Ksh v.93u+ 2012-08-01. All scripts were executed completely unmodified, using environment variables, loops, and other shell

Tab. 2: Correctness results. Running the POSIX test suite on Bash and PASH-JIT. Tests are grouped in rows by theme. Columns contain the group name, total tests, non-applicable tests, and passing tests for PASH-JIT and Bash.

	Test Suite	Tests	Untested	PASH-JIT	Bash
1	Parsing	38	5	33/33	33/33
2	Expansion	83	8	71/75	71/75
3	Errors	38	3	26/35	27/35
4	Commands and redirects	99	2	96/97	96/97
5	Subshells and pipelines	56	7	46/49	46/49
6	Builtins	113	40	60/73	61/73
7	Special cases	67	21	42/46	42/46

constructs. To minimize statistical non-determinism, we host our experimental infrastructure on our own premises, avoid sharing with other research groups, and repeat the experiments several times noting imperceptible variance.

7.1 Correctness

We evaluate the correctness of PASH-JIT across all benchmarks from Tab. 1 by checking that PASH-JIT's stdout and exit status are equivalent to the ones produced from Bash. The output is over 650 million lines (18GB), taken from 82 scripts, in all of which PASH-JIT's output and exit status are correct. To increase our confidence on correctness, we use the POSIX shell test suite with both Bash and PASH-JIT.

Benchmarks: The POSIX test suite is a thorough evaluation of shell behavior, comprising 1007 'assertions' evaluated using 494 distinct, assertion-numbered test cases over 29k LOC of shell scripts (plus library support). We exclude (a) 78 test cases because they test the platform (*e.g.*, locales) rather than the shell, and (b) 8 cases because they test interactivity, which is out of scope for PASH-JIT (§1). These leave a total of 408 runnable test cases. The test cases use a mix of shell language features (*e.g.*, redirection, pipes), builtin commands (*e.g.*, `set`, `echo`), and standard UNIX utilities (*e.g.*, `printf`, `grep`). The POSIX suite tests many corner cases of shell behavior—*e.g.*, that aliases ending in space continue alias expansion (Assertion no. 284), that pipelines take precedence over redirections in their constituent commands (no. 454), or that **`return`** in a `trap` action restores the previous command's exit status (no. 651)—totaling several thousand behaviors. The exact number of 'tests' is hard to quantify: some test cases check a single behavior (*e.g.*, expanding an unset variable under `set -u`); others check hundreds (*e.g.*, many different characters escape properly; many different arithmetic expressions evaluate correctly).

Results: PASH-JIT overwhelmingly agrees with Bash (Tab. 2). PASH-JIT passes 374 and fails 34 POSIX tests, while Bash passes 376 and fails 32 POSIX tests. PASH-JIT diverges from Bash on the test cases for a mere 2 tests (no. 430 and 691) where Bash passes but PASH-JIT fails. These two failures concern the ranges of non-zero exit status and are in fact due to an unusual inconsistency in Bash itself (see "Discussion", below).

When running the test suite, PASH-JIT invokes the compiler a total of 3304 times, each for a different potentially optimizable fragment; 713 (20%) of those invocations successfully compile, *i.e.*, PASH-JIT generates and runs parallel code. Successful compilation does not necessarily translate to a speedup on individual tests, though: the POSIX suite tends to test with small scripts, so the compiled fragments contain very little computation—not much for PASH-JIT to optimize.

Discussion: PASH-JIT diverges from Bash in two cases only in the exit status returned. Both PASH-JIT and Bash exit with an error: Bash returns 1, and PASH-JIT returns 127. For the two failing cases, POSIX mandates (since 2008) that the exit status be between 1–125, making PASH-JIT's behavior incorrect. Why does PASH-JIT produce a different status?

Bash is inconsistent when called with the `-c` flag. Contrary to most other shells (*i.e.*, dash, ksh, mksh, posh, sash, Smoosh, yash, zsh), Bash is the only shell that, when failing during `-c` invocations, exits with 127—*i.e.*, outside the POSIX-mandated range. When PASH-JIT invokes the underlying Bash interpreter using `-c` in order to set `$0`, it receives and propagates an exit status that does not comply with POSIX. The rest of the Bash failing tests are caused by various subtleties; it is not clear which failures are 'true bugs' and which are considered desirable divergences from the spec. Greenberg and Blatt [24] discuss how implementations diverge from the POSIX spec. PASH-JIT mirrors the behavior of Bash in all those cases.

To put the number of diverging tests of PASH-JIT and Bash into perspective, we note that other production shells fail in significantly greater numbers: dash passes 3 tests that Bash fails and fails 20 that Bash passes; ksh passes 2 tests that Bash fails and fails 20 that Bash passes; and zsh cannot run the test suite at all. These results combined show that, in practice, PASH-JIT is virtually indistinguishable from its underlying shell interpreter on POSIX features.

7.2 Performance

We evaluate PASH-JIT's performance on 12 sets of real-world shell scripts taken from a variety of sources (Tab. 1, rows 2–13), totalling 82 shell scripts and 1015 LOC.

Benchmarks: Classics and Unix50 contain classic and recent (c. 2019) scripts making heavy use of UNIX and Linux built-in commands. COVID-mts contains four scripts used to analyze real telemetry data from mass-transit schedules during a large metropolitan area's COVID-19 response. NLP contains several scripts from UNIX-for-poets, a tutorial for developing programs for natural-language processing out of UNIX and Linux utilities. AvgTemp contains a large script downloading and processing multi-year temperature data across the US. WebIndex is a large multi-stage script for web crawling and

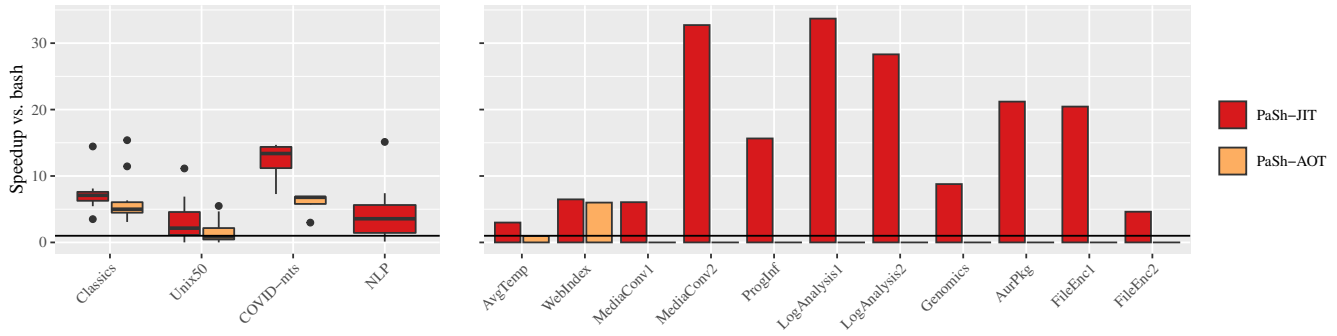


Fig. 6: PASH-JIT Performance. PASH-JIT speedup (*vs.* PASH-AOT whenever possible) over Bash for Tab. 1 rows 2–5 (left, box) and 6–13 (right, bar) (*Cf.*§7.2).

indexing, using a variety of third-party and built-in utilities. MediaConv contains two scripts that process, transform, and compress video and audio files. ProgInf contains a script that downloads JavaScript packages from the npm registry and applies a security-oriented static program analysis. LogAnalysis contains two scripts that apply typical system-administration and network-traffic analyses over log files. Genomics contains a script that processes next-generation sequencing data for the purposes of diagnostic virology. AurPkg contains the main script that compiles, builds, and packages software for the AUR Linux distribution. Finally, FileEnc contains long aliases that encrypt and compress files.

Results: PASH-JIT surpasses PASH-AOT's speedups (*vs.* Bash) on existing benchmarks and extends speedups to new ones (Fig. 6). Box-plots show results for multi-benchmark suites (Tab. 1, rows 2–5) and bars for individual scripts (Tab. 1, rows 5–13). PASH-JIT can run several more scripts than PASH-AOT (for which performance bars are set to 0). Across all benchmarks, PASH-JIT achieves an average speedup of 5.86× (*vs.* 2.9× for PASH-AOT) and a maximum speedup of 33.7× (*vs.* 15.38× for PASH-AOT).

A few scripts exhibit slowdowns when compiler startup, runtime, and parallelization overheads (splitting, merging) start dominating. PASH-JIT decelerates 14 scripts; PASH-AOT decelerates 20 scripts—and cannot run 30 additional scripts that PASH-JIT parallelizes. The scripts that PASH-JIT decelerates either have short sequential running times (8ms–10s) or have very short-running fragments in tight loops (*e.g.*, 1K iterations, 14ms per iteration). For example, PASH-JIT decelerates Unix50's `20.sh` (Bash: 8ms; PASH-JIT: 1.3s) and NLP's `no-vowel.sh` (Bash: 14s; PASH-JIT: 0.24×), on which PASH-AOT cannot operate.

Discussion: PASH-JIT is faster than PASH-AOT on all suites 2–5 (w.r.t. average) and on all individual benchmarks 5–13, often by a significant margin (3.1×).

PASH-JIT speeds up many scripts PASH-AOT cannot, as PASH-AOT's ahead-of-time parallelization cannot reason about the shell's dynamic features. PASH-AOT offers no speedup on the NLP suite, nor on any individual scripts except for AvgTemp and WebIndex.

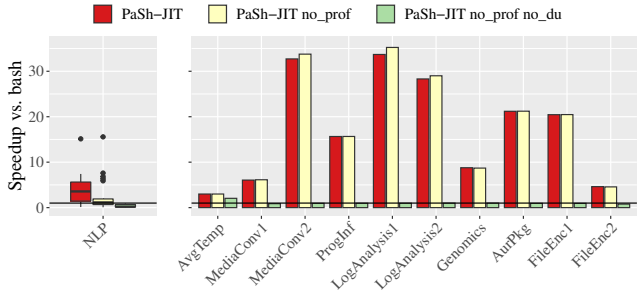


Fig. 7: PASH-JIT Dynamic Optimizations. PASH-JIT speedup over Bash when toggling profile-driven compiler configuration and dependency untangling for Tab. 1 row 5 (left, box) and 6, 8–13 (right, bar) (*Cf.*§7.3).

Compared to Bash, PASH-JIT is faster (or at least as good) in all cases, except when the given script is very short-running (*e.g.*, `unix50-20.sh`), or with a tight loop with a very short-running body (*e.g.*, `nlp-no-vowel.sh`).

7.3 Further Microbenchmarks

This section zooms into the benefits of PASH-JIT's optimizations targeting dependency untangling, profile-driven compiler configuration, commutativity analysis, and JIT engine overheads.

Dynamic optimizations: To better understand the benefits of dependency untangling and profile-driven compiler configuration (CC), we use benchmarks that have sequences of statements—*e.g.*, some form of sequential composition or `for`-loops: rows 5, 6, 8–13 from Tab. 1. One-line scripts such as Unix50 and WebIndex feature single pipelines and thus cannot benefit from any inter-region optimizations.

Across all scripts and compared to Bash, PASH-JIT achieves a speedup of 8.17×. PASH-JIT without profile-driven CC achieves 7.58×, and additionally without dependency untangling 0.55× (Fig. 7). The 0.55× slowdown is due to limited intra-region parallelization in these benchmarks.

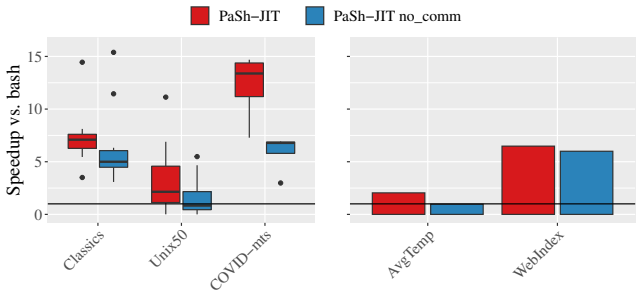


Fig. 8: PASH-JIT Commutativity Awareness. PASH-JIT speedup over Bash when toggling commutativity awareness for Tab. 1 rows 2–4 (left, box) and 6, 7 (right, bar) (*Cf.*§7.3).

Profile-driven CC may slightly reduce speedup in highly parallelizable scripts, because it explores lower parallelization degrees.

Commutativity awareness: To evaluate the benefits of commutativity-related optimizations, we focus on all scripts with intra-region parallelization potential: Classics, Unix50, COVID-mts, AvgTemp, and WebIndex; the performance of the rest is affected negligibly by changes to single-region transformations. We disable all dynamic optimizations to isolate the benefits of commutativity, and compare with the sequential Bash baseline.

Commutativity-aware PASH-JIT achieves an average speedup of 4.52× and a maximum of 14.68× (Fig. 8). Without commutativity-related optimizations, PASH-JIT achieves an average speedup of 3.72× and a maximum of 15.38×. Commutativity improves the average case but not cases that already see high speedups, as these (1) have negligible overheads coming from input reading—most overheads come due to line processing—and (2) commutativity extensions add some overhead due to the `c_wrap` primitive.

JIT engine overhead: To evaluate the benefits of PASH-JIT's runtime optimizations, we design a worst-case parallelization benchmark: a script that contains a **`for`** loop that performs 100 iterations of `echo hi`. A tight loop with a minimal-overhead body emphasizes the JIT engine overheads by allowing no parallelization gains. The table on the right shows the run-time performance of four PASH-JIT configurations compared to Bash: (1) PASH-JIT without custom expansion, compilation server, and dynamic optimizations, (2) PASH-JIT without compilation server, and dynamic optimizations, (3) PASH-JIT without the dynamic optimizations, and (4) the complete PASH-JIT. PASH-JIT's runtime optimizations (custom expansion, compilation server, and dependence untangling) improve performance by 12× (over the `-esd` configuration without them). As `echo hi` writes to stdout, dependence untangling does not manage to run it in parallel, and thus its benefit is only due to pipelining. Even then, PASH-JIT's JIT engine overhead is not negligible (about 47ms per JIT invocation), as it needs to save the state and invoke the compiler for every iteration of the loop body.

Config.	Time (s)
Bash	0.008
PASH-JIT -esd	59.334
PASH-JIT -sd	15.376
PASH-JIT -d	6.124
PASH-JIT	4.708

8 Related Work

Parallel shell scripting: Recent work addresses significant challenges related to automatic shell script parallelization. POSH [52] and PASH-AOT [63] are mostly-automated ahead-of-time shell-script parallelization systems; as described earlier, these systems focus on fully expanded shell pipelines that do not make use of dynamic features. Recent work explored an order-aware dataflow model as a foundation for modeling the transformations these systems perform and proving them correct [28]. To enable divide-and-conquer parallelism, KumQuat [55] proposes a program-synthesis technique for generating aggregators for black-box commands.

PASH-JIT builds on all this prior work, addressing fundamental limitations in static, ahead-of-time parallelization: AOT approaches apply to a very small subset of real shell scripts. By opting for just-in-time parallelization, PASH-JIT achieves parallel script behavior that is practically indistinguishable from the sequential execution—and ample opportunities for additional acceleration.

Other work on shell script parallelization either requires manual effort or is applicable to a smaller subset of scripts than our work. Such work includes: utilities like `qsub` [19], SLURM [66], and `parallel` [58]; shells with non-linear pipe topologies [17, 40, 56]; and using the shell itself as a DSL for concurrency [22].

Unix-related parallelization: There has been a significant body of work on parallel (and distributed) UNIX and UNIX-like environments [4, 44, 47], including shell-oriented efforts such as Plan9's `rc` [49]. Contrary to PASH-JIT, these systems did not (aim to) offer full compatibility with the sequential UNIX shell. They also focused on systems-level and program-runtime support, rather than automated program analyses and transformations.

Just-in-time compilation: Just-in-time compilation has been studied for long time [3], mainly in two contexts: (1) as a compilation technique for interpreted languages such as JavaScript [20], where critical type information is unavailable prior to execution; and (2) as a performance optimization over ahead-of-time compilation, allowing for specialization [30, 60], loop unrolling and function inlining [9, 50], and other profile-guided optimizations [34, 46]. PASH-JIT draws inspiration from work in both contexts—resolving unavailable dynamic information at run-time and performing additional optimizations. It also leverages the optimistic compilation technique employed commonly by just-in-time compilers: when it fails to compile (parallelize), it simply runs the original fragment using the shell interpreter as a fallback option.

PASH-JIT differs from most JITs, dealing with different challenges: it operates at a higher level of abstraction, in a unique programming environment with no single unified runtime.

PASH-JIT also draws inspiration from staged compilation [14] and partial evaluation [32]. These techniques perform some compilation ahead-of-time, waiting for the runtime to specialize and further optimize when there is more information about the environment of the target program and how it is used.

Parallelization in other contexts: More general parallelization support can be grouped into two categories: languages and tools. One approach to parallelization support is to use tools that requires writing in a new higher-level programming language [18, 21, 36] or a dataflow-based model embedded in an existing language [5, 12, 16, 45, 57, 67]. These tools usually offer automation, but require re-expressing existing computations in domain-specific programming models; PASH-JIT operates on completely unmodified POSIX shell scripts that use unusual features and obscure corner cases.

Another approach to parallelization support uses tools that provide automatic parallelization for standard sequential code, requiring no program modifications but often posing limitations with respect to the granularity of the parallelism that they can extract. The general approach started with explicit `DOALL` and `DOACROSS` annotations [10, 38], continuing with analysis-based compilers [27, 48, 54], and more recent work using profiling-guided speculation [1, 31, 35, 42, 61]. PASH-JIT draws inspiration from this line of work: it does not require manual modification to user code, and it leverages run-time information to optimize and parallelize user scripts. Existing tools work on imperative code with memory accesses, but PASH-JIT works at a higher level of abstraction: commands that affect the file system and the broader executing environment.

Shell correctness and POSIX compliance: Smoosh [24] offers a formalized, executable reference semantics for the POSIX shell, aiming to address subtleties in the standard [2]. PASH-JIT leverages Smoosh to identify and resolve issues in its JIT engine (§4) and to guide its early expansion routine (§5.2). It also builds on Smoosh's analysis to leverage the POSIX test suite for characterizing shell behavior.

PASH-JIT reimplements Smoosh's `libdash` [23], which presents dash's parser as a library (§3.3). We chose `libdash` over Morbig [53] because (1) `libdash` reuses dash's production-grade parser, and (2) `libdash` supports line-oriented input, but Morbig is strictly ahead-of-time.

Resurgence of shell research: Recent shell research [24, 25, 39, 43, 52, 55, 56, 63] highlights renewed interest in shell scripting both as a vehicle for impactful research and as a target worthy of scientific attention. We see PASH-JIT as a natural continuation of the insights and research behind recent shell-script parallelization systems [25, 28, 52, 63], allowing other researchers to leverage PASH-JIT's POSIX-compliant high-performance just-in-time compilation in their future work.

9 Discussion & Conclusion

The shell provides a dynamic programming language with complex evaluation-and-expansion semantics and ubiquitous side-effects—effects that interact with the entire UNIX system similar to how a conventional programming language interacts with its runtime environment. The benefits of just-in-time compilation for dynamic languages are clear, and PASH-JIT is the first JIT compiler that targets challenges unique in the UNIX shell ecosystem. PASH-JIT forms a promising drop-in shebang replacement: its POSIX compliance rivals shells in widespread use; and its performance benefits go well beyond the state of the art.

Interactivity: PASH-JIT's design goals (§1) do not include interactivity; an interactive shell switches between consuming its input (shell commands) and redirecting it to its executing commands—challenging for PASH-JIT's loose coupling. Furthermore, avoiding shell modifications leads to additional runtime overhead (since the state of the shell has to be reflected upon and is not accessible with a single dereference). Adding robust support for interactivity and improving runtime overhead would likely require a more intrusive design, *e.g.*, altering Bash's source and interposing directly. However, such a design would make PASH-JIT Bash-specific, requiring users to install a new shell, and would significantly complicate the engineering and maintenance effort involved.

Expansion: Some of PASH-JIT's expansion behaves in a way not exactly as specified by POSIX, although we conjecture (and our evaluation confirms, §7) it is safe. For example, pipelines are supposed to expand each component in its own subshell (though the last component may run in the outer shell, depending on a shell's implementation choices). PASH-JIT's expansion operates on each component of the pipeline early; each component uses its own copy of the shell environment, to simulate the subshells. We haven not proved these early expansions sound, and it would be interesting future work to pursue that, *e.g.*, by using Smoosh's semantics.

Command annotations: PASH-JIT's performance benefits depend on the existence of command parallelizability annotations. The annotations used by PASH-JIT depend on the PASH-AOT annotation library [63], which includes many commands in the POSIX and GNU Coreutils sets. Apart from commands in these sets, a script may contain other commands—for which PASH-JIT will lack annotations and thus will not attempt to parallelize to maintain soundness (§1). To better harness PASH-JIT parallelization in their scripts, users can: (1) opt for more restricted, rather than more general, utilities with more constrained and thus parallelizable behaviors (*e.g.*, use `cut` rather than `awk` when projecting columns, as `awk` programs are not parallelizable in general); or (2) add

their own annotations for custom commands to inform PASH-JIT on how to parallelize them.

Enabling other analyses: Even though PASH-JIT is mainly focused on parallelization, its just-in-time structure is not limited to it. By slightly modifying the preprocessor and by replacing the compilation server logic, PASH-JIT can be made to perform different types of analyses and transformations, while maintaining its benefits—compliance with the underlying shell, loose coupling, and low runtime overheads. This enables exciting avenues of future tooling and support for the shell, like incremental execution, automatic distribution, and safety monitoring.

Conclusion: Fundamentally, PASH-JIT shows that it is possible to build a just-in-time shell-script parallelization infrastructure that is substantially faster and more applicable than prior work, is loosely coupled, and addresses critical challenges associated with the shell ecosystem's polyglot runtime environment. But also, PASH-JIT is not a toy: it enables other researchers to use a production-grade POSIX-compliant shell compiler for impactful future work.

Acknowledgements: We would like to thank Achilles Benetopoulos, Ben Karel, Caleb Stanford, the OSDI 2022 reviewers, and our shepherd, Robert Soulé, for discussions and feedback that helped improve the presentation of the paper; the OSDI 2022 AE reviewers for feedback that improved this paper's artifact; the participants of UCSC's LSD seminar for early discussions on dependency untangling; and the open-source developers who have contributed to PASH. This material is based upon work supported by DARPA contract no. HR00112020013 and no. HR001120C0191, and NSF awards CCF 1763514 and 2008096.

References

[1] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 351–367, 2020.

[2] The Austin Group. POSIX.1 2017: The Open Group Base Specifications Issue 7 (IEEE Std 1003.1-2008), 2018.

[3] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[4] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4):361–372, 1998.

[5] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: a C++ template library for high performance stream parallel processing. *The International Journal of High Performance Computing Applications*, 31(5):391–404, 2017.

[6] Jon Bentley. Programming pearls: A spelling checker. *Commun. ACM*, 28(5):456–462, May 1985.

[7] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: A literate program. *Commun. ACM*, 29(6):471–483, June 1986.

[8] Pawan Bhandari. Solutions to unixgame.io, 2020. Accessed: 2020-04-14.

[9] Carl Friedrich Bolz. *Meta-tracing just-in-time compilation for RPython*. PhD thesis, Universitäts-und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2014.

[10] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 162–175, New York, NY, USA, 1986. ACM.

[11] Enrico Cappellini, Frido Welker, Luca Pandolfi, Jazmín Ramos-Madrigal, Diana Samodova, Patrick L Rüther, Anna K Fotakis, David Lyon, J Víctor Moreno-Mayar, Maia Bukhsianidze, et al. Early pleistocene enamel proteome from dmanisi resolves stephanorhinus phylogeny. *Nature*, 574(7776):103–107, 2019.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[13] Armando Cerna. Pacaur building script.

[14] Craig Chambers. Staged compilation. *ACM SIGPLAN Notices*, 37(3):1–8, 2002.

[15] Kenneth Ward Church. Unix™for poets. *Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods*, 1994.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[17] Tom Duff. Rc—a shell for plan 9 and unix systems. *AUUGN*, 12(1):75, 1990.

[18] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[19] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36. IEEE, 2001.

[20] Google. V8 javascript engine. https://developers.google.com/v8/.

[21] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. A stream compiler for communication-exposed architectures. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 291–303. ACM, 2002.

[22] Michael Greenberg. The posix shell is an interactive dsl for concurrency. https://cs.pomona.edu/~michael/papers/dsldi2018.pdf, 2018.

[23] Michael Greenberg. libdash. https://github.com/mgree/libdash, 2019. [Online; accessed December 6, 2021].

[24] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the POSIX shell: Smoosh: the symbolic, mechanized, observable, operational shell. *Proc. ACM Program. Lang.*, 4(POPL):43:1–43:30, January 2020.

[25] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. Unix shell programming: The next 50 years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 104–111, New York, NY, USA, 2021. Association for Computing Machinery.

[26] The Open Group. Posix. https://pubs.opengroup.org/onlinepubs/9699919799/, 2018. [Online; accessed November 22, 2019].

[27] Mary W Hall, Jennifer M Anderson, Saman P. Amarasinghe, Brian R Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[28] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. An order-aware dataflow model for parallel unix pipelines. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[29] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[30] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 294–310, 2000.

[31] Nick P Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I August. Speculative separation for privatization and reductions. *ACM SIGPLAN Notices*, 47(6):359–370, 2012.

[32] Neil D Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

[33] Dan Jurafsky. Unix for poets, 2017.

[34] Konstantinos Kallas and Konstantinos Sagonas. Hiperjit: A profile-driven just-in-time compiler for erlang. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, pages 25–36, 2018.

[35] Hanjun Kim, Nick P Johnson, Jae W Lee, Scott A Mahlke, and David I August. Automatic speculative doall for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 94–103, 2012.

[36] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. Optimistic parallelism requires abstractions. *ACM SIGPLAN Notices*, 42(6):211–222, 2007.

[37] Nokia Bell Labs. The unix game—solve puzzles using unix pipes, 2019. Accessed: 2020-03-05.

[38] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

[39] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track*, pages 9–15, 2021.

[40] Chris McDonald and Trevor I Dix. Support for graphs of processes in a command interpreter. *Software: Practice and Experience*, 18(10):1011–1016, 1988.

[41] Malcolm D McIlroy, Elliot N Pinson, and Berkley A Tague. Unix time-sharing system: Foreword. *Bell System Technical Journal*, 57(6):1899–1904, 1978.

[42] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *ACM Sigplan Notices*, 44(6):166–176, 2009.

[43] Jürgen Cito Michael Schröder. An empirical investigation of command-line customization. *arXiv preprint arXiv:2012.10206*, 2020.

[44] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[45] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[46] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

[47] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.

[48] David A Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin. Polaris: A new-generation parallelizing compiler for mpps. In *In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign*, 1993.

[49] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*, pages 1–9, 1990.

[50] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, 1998.

[51] Jon Puritz. Bio594: Using genomic techniques to examine the evolution of populations, 2019. Accessed: 2020-10-05.

[52] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: A data-aware shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 617–631, 2020.

[53] Yann Régis-Gianas, Nicolas Jeannerod, and Ralf Treinen. Morbig: A Static Parser for POSIX Shell. In *Software Language Engineering (SLE)*, Boston, United States, November 2018.

[54] Martin C Rinard and Pedro C Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, 1997.

[55] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. Automatic synthesis of parallel unix commands and pipelines with kumquat. corr abs/2012.15443 (2021). *arXiv preprint arXiv:2012.15443*, 2021.

[56] Diomidis Spinellis and Marios Fragkoulis. Extending unix pipelines to dags. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[57] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, page 9–16, New York, NY, USA, 2011. Association for Computing Machinery.

[58] Ole Tange. Gnu parallel—the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[59] Dave Taylor. *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*. No Starch Press, 2004.

[60] Scott Thibault, Charles Consel, Julia L Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[61] Chen Tian, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 62–73, 2010.

[62] Eleftheria Tsaliki and Diomidis Spinellis. The real statistics of buses in Athens. https://bit.ly/3s112R5, 2021.

[63] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[64] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on node.js via RWX-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1821–1838, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.

[66] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[67] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

A Artifact Appendix

The structure of this section mirrors the artifact evaluation process. It is a shorter version of the README available in the frozen `osdi22-ae` branch of PaSh's GitHub repository. At a glance:

- Artifact available: Relevant links pointing to online resources.
- Artifact functional: Documentation, completeness with respect to the claims in paper, and exercisability.
- Results reproducible: Instructions for reproducing correctness (§7.1), performance (§7.2), and microbenchmark (§7.3) results.

A.1 Artifact available

The implementation described in this paper has been incorporated into PaSh, an MIT-licensed open-source software available by the Linux Foundation. Below are some relevant links:

- PaSh is permanently hosted on the GitHub binpash organization.
- The PaSh website is available at binpa.sh and https://binpash.github.io/web/.
- PaSh has joined the Linux Foundation and is available via Dockerhub.
- PaSh developers hang out on the pash-discuss mailing list and discord.

PaSh is developed actively, forms the foundation of further research on the shell, and has received open-source contributions from developers outside the core development team.

A.2 Artifact functional

Fig. 1 gives an overview of the interaction between different components and the correspondence of system components to sections. Below we provide links to the source code implementing them. Note that at the time of writing the terminology in the code is somewhat different from the one presented in the paper; we hope to align the code with the paper soon.

- Preprocessor (§3.1 and 3.2): The preprocessor uses the parser (below) to instrument the script AST with calls to the JIT Engine.
- Parsing library (§3.3): The parsing library contains Python bindings for the dash parser and a complete unparser implementation.
- JIT engine (§4): The JIT engine transitions between shell and PaSh mode and interacts with the parallelizing compilation server (below).
- Parallelizing compilation server (§5): The parallelizing compilation server handles compilation requests for parallelizing regions of the script. The server contains the following subcomponents: (i) the early expansion component (§5.2); (ii) the dependency untangling component (§5.3), enabled with `--parallel_pipelines`; and (iii) the profile-driven configuration component (§5.4), enabled with `--profile-driven`.
- Commutativity awareness (§6): It consists of (i) annotations indicating whether a command is commutative (*e.g.*, `sort`) and (ii) dataflow nodes for orchestrating commutativity-aware parallelization—*e.g.*, `c-split`, `c-wrap`, `c-strip`, and `c-merge`.
- The paper also claims that the core of the server has been modeled and verified using SPIN. The modeling of the dependency untangling algorithm in Promela (SPIN's language) can be found in algorithm.pml. The model captures compilation requests of regions with non-deterministic read/write dependencies, and ensures that no two regions with dependencies are running together, while also ensuring that both the server and the engine eventually terminate.

A.3 Results reproducible

The paper contains three classes of experiments, focusing on:

- correctness/compatibility, using the entire POSIX test suite as well as additional scripts (§7.1).
- performance gains achieved by PaSh-JIT's parallelization, evaluated using a variety of benchmarks and workloads (§7.2).
- PaSh-JIT-internal overheads and associated optimizations (§7.3).

Links to these can be found in the relevant section of the artifact README. The POSIX test suite is from the Open Standards Group and thus cannot be shared outside the Docker container on the machine shared with the AEC reviewers. These tests run via CI on every commit on the PaSh project. Instructions to verify the dependency untangling algorithm can be found here.

Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android

Yu Luo
University of Toronto

Kirk Rodrigues
University of Toronto

Cuiqin Li
Huawei Technologies Co., Ltd.

Feng Zhang
Huawei Technologies Co., Ltd.

Lijin Jiang
Huawei Technologies Co., Ltd.

Bing Xia
Huawei Technologies Co., Ltd.

David Lion
University of Toronto

Ding Yuan
University of Toronto

Abstract

Hubble is a method-tracing system shipped on all supported and upcoming Android devices manufactured by Huawei, in order to aid in debugging performance problems. Hubble instruments every non-inlined bytecode method's entry and exit to record the method's name and a timestamp. Instead of persisting all data, trace points are recorded into an in-memory ring buffer where older data is constantly overwritten. This data is only persisted when a performance problem is detected, giving engineers access to invaluable, detailed runtime data Just-In-Time before the detected anomaly. Hubble is highly efficient, with its tracing inducing negligible overhead in real-world usage and each trace point taking less than one nanosecond in our microbenchmark. Hubble significantly eases the debugging of user-experienced performance problems and has enabled engineers to quickly resolve many bug tickets that were open for months before Hubble was available.

1 Introduction

Today, Android devices are pervasive and tightly integrated into people's daily lives, yet users still experience performance problems when using these devices. Unlike Apple's iOS and iPhone, the Android platform is far from a tightly-coupled monolithic ecosystem—the hardware (manufactured by OEMs), infrastructure system software (maintained by Google and customized by OEMs), and applications are provided by different parties, and all layers are released in a rapid yet uncoordinated development cycle. This open platform makes testing enough combinations of hardware, systems software, and applications particularly challenging. Thus, many of the performance bugs that escape current testing practices are intermittent, manifesting across multiple components maintained by different entities.

When end users experience an issue, it is often *systems* vendors that shoulder the blame, before the root cause is exposed [49]. This is particularly true for Android given its huge user base, many of whom are not tech-savvy. When such users experience an intermittent performance problem, they quickly assume that their device is at fault, simply because they could not immediately reproduce the issue on another device. However, the root cause could be in the application itself, only triggered under specific conditions or inputs. To combat these assumptions, device vendors are forced to devote ample engineering and support resources to these issues.

Yet, diagnosing performance problems that occur on a user's device is extremely challenging, owing to a lack of sufficient runtime information. While approaches like Windows Error Reporting (WER) [21] are widely adopted, they can only record runtime information *after* a problem is detected. Oftentimes this is too late, as it misses crucial information just before and during the problem. This is exacerbated for performance problems, especially intermittent ones, because the issue may vanish after being detected, before recording starts. Indeed, the primary use of WER is not to record enough information to debug an issue, but to collect error statistics that are then used to prioritize debugging effort.

Recording debugging information *before* the problem occurs is challenging. We cannot accurately predict when a problem will occur, so the only option is to continuously trace the system during *normal* execution. However, overhead is a concern. Unlike servers, mobile devices are heavily resource-constrained and their workloads are overwhelmingly interactive. Sampling-based profiling tools are available, but their trade-off between informativeness and performance is poor. Non-sampling-based profiling tools, on the other hand, are too heavyweight for continuous tracing. For example, existing profiling tools on Android like Systrace [25] and Android Studio's CPU Profiler [24] can trace every method call of an application. However, enabling this type of tracing noticeably slows down an application, sometimes by more than 10×, which is unsuitable for continuous use in production. Individual applications may implement their own in-app tracing [18,23,46], but such traces are typically only available to those applications themselves.

As a result, problems reported to Android device vendors typically only include system logs, sampled statistical metrics,

Design	Unnoticeable	No src.	Maintain	big. LITTLE
Instrumentation via JIT	✓	✓		
Ring-buffer & encoding	✓			
Hand-optimized asm	✓		✓	✓
Lock-free control	✓			

Table 1: Hubble's designs and the requirements they satisfy. The headings for the requirements are truncated as follows: "Unnoticeable" refers to having unnoticeable overhead. "No src." refers to not requiring source code. "Maintain" refers to being maintainable. "big.LITTLE" refers to supporting both big and little cores.

sparse Systrace traces, and details recorded *after* a problem has occurred, like the device model, application name, and symptom. Most times, this is not enough to be useful in debugging intermittent performance problems, and engineers are left "debugging in the dark." Consequently, many bug tickets are left open for months without any hope of resolution. Worse yet, many bugs cannot even be properly *triaged*, and after rounds of finger-pointing, it is often the low-level system engineers that bite the bullet.

1.1 Challenges and Opportunities

Therefore, a *production* tracing system that can provide fine-grained observability is desperately needed. However, continuous tracing in production is challenging; it needs to satisfy a number of stringent requirements. First, the *worst-case* overhead must be unnoticeable (it cannot exceed 3% or increase the number of performance regressions throughout the deployment cycle), regardless of whether the application is running on the powerful (big) or weaker (little) cores in ARM's big.LITTLE architecture. In addition, the tool should trace applications without access to their source. Finally, it needs to be easy to maintain, and easy to merge with every new (and often feature-breaking) Android release.

These goals and constraints are stricter than what is offered by existing solutions. For instance, while record and replay (R&R) can faithfully replay the entire execution, we are not aware of any R&R system that can achieve *worst-case* overhead below 3%. In fact, most literature [30,33,35,57] emphasizes the *average* overhead; for *production* tracing tools on Android devices, engineers are primarily concerned with the worst-case instead of the average. In addition, R&R techniques typically require deep integration with the Android runtime which means that they cannot be easily maintained.

Another challenge offered by the Android runtime environment is the semantic gap between an application written in a high-level language (Java) and its native execution, which renders a rich set of system profiling tools such as gprof [27] ineffective without the runtime's support. When applied to runtime workloads, these profilers only profile the runtime's execution instead of the applications running on top of it. For example, applying gprof to a runtime workload only provides the call graph of the runtime itself (including the interpreter, GC, and JIT-compiled code), instead of the call graph of the Java application.

Android [26] and other runtimes [7] can output symbol information during execution so that system profiling tools can be applied to profile language-level executions. This approach does not completely close the semantic gap for a few reasons. First, each profiling tool must support using these symbols; currently only the sampling-based perf [39] tool supports using the symbols, and only for JIT-compiled code. Android extended and integrated perf such that it can also profile the interpreter's execution at the language-level [26]. In addition, perf expects every symbol to have a unique memory address, which is not always true; for instance, the runtime may update JIT-compiled code with application hot-patching or recompilation based on new profiling information, thus unloading old mapped code and reusing the page [26].

Yet, the runtime environment also presents a unique opportunity: trace points can be embedded and removed transparently by the runtime without modifying the application's source. This opportunity remains under-exploited despite the popularity of managed languages (the five most popular languages on GitHub in 2021 were runtime languages). To the best of our knowledge, none of the existing language runtimes offer detailed tracing tools that can be used continuously in production. For example, the OpenJDK JVM provides a powerful JVMTI debugging interface that can embed breakpoints in applications. However, this means that execution has to be deoptimized and run in the interpreter (rather than JIT compiled). Therefore, it is mostly suitable for use in development environments. Many runtimes also provide sampling-based profiling features that show "hot" code paths, but none provide continuous method-level tracing suitable for production.

1.2 Contributions

This paper presents the design and implementation of Hubble that satisfies the aforementioned goals. Hubble can capture most method entry and exit points of any application's threads, just-in-time before a failure. We designed Hubble by combining several well-known techniques in a novel way that takes advantage of the Android platform. Table 1 shows Hubble's major designs and the requirements they satisfy.

First, Android applications are typically downloaded as bytecode and then either compiled or interpreted on the device; Hubble leverages this runtime environment to automatically embed its tracing logic into the compiled binary or interpreted logic. This enables efficient tracing, as the tracing logic can be inlined into the application, avoiding more expensive trampolines (i.e., jumps in control flow) that are common in other tracing tools. In addition, this means that Hubble is a purely black-box approach that does not depend on the application's source code.

In addition, Hubble writes trace points to an in-memory ring buffer that is only flushed when a problem is detected. This allows it to run continuously and capture information just-in-time leading up to a failure. By designing a concise, variable-length encoding, such that most trace points occupy eight bytes, a small (32MB) ring buffer is enough to capture sufficient debugging information.

Third, Hubble's performance-sensitive instrumentation logic is written in assembly. This ensures that performance is optimal even on a device's low-power (little) cores, which cannot perform out-of-order execution or have small instruction reordering buffers. In addition, this decouples Hubble from the Android compiler's compilation flow, so it avoids having the compiler affect the correctness of the tracing logic, and eases maintainability.

Finally, Hubble avoids using expensive synchronization primitives [14] in two ways: threads write trace points to thread-local buffers, avoiding inter-thread synchronization; and, Hubble communicates with these threads by using a purpose-built lock-free synchronization protocol.

The end result is a highly efficient method-tracing system sufficient for debugging intermittent performance bugs. In our microbenchmarks, each trace point costs less than one nanosecond for nearly empty methods, and tracing overheads are quickly amortized when methods perform meaningful operations. Hubble's tracing overhead is also unnoticeable in Huawei's continuous-integration performance testing infrastructure, which includes a variety of workloads and devices. Hubble's memory overhead is approximately 64 MB by default, accounting for two 32 MB ring buffers. As of 2021, Huawei's lower-end smartphones have at least 4 GB of RAM, while higher-end ones can have up to 12 GB. Therefore, Hubble's memory overhead is less than 2%.

Hubble also strives to protect user privacy. Similar to existing error reporting systems such as WER [21], MacOS [2] and Mozilla [34] crash reports, Hubble's traces are only collected with user consent. However, these other systems collect a minidump of the memory image, whereas Hubble's traces are far less sensitive: they only consist of method names and timestamps and do not contain any variable values.

Hubble has been integrated into Huawei's core Android OS codebase, deployed across a wide range of smartphone and tablet product lines, since August, 2020. Older devices may receive Hubble's functionalities via an over-the-air OS update. Since deployment, Hubble has significantly eased the debugging of intermittent performance problems. In fact, engineers were able to quickly resolve many performance problems that remained unresolved for months.

This paper makes the following contributions:

- The design and implementation of Hubble, a highly efficient method tracing subsystem for Android, that satisfies a set of unique, practical constraints, some of which are rarely mentioned by existing literature.
- Integration of Hubble's traces with existing debugging tools, like Perfetto [40] which can show call charts. This significantly improved the trace's utility, where developers can cross-examine Hubble traces with other runtime data.
- Case studies on how Hubble diagnoses real-world performance bugs which cannot be resolved without it.

Hubble also has the following limitations. First, it can only embed tracing logic into executions that go through the Android compiler or interpreter (from bytecode); Hubble cannot trace native libraries like those invoked through the Java Native Interface (JNI). In addition, Hubble's trace buffer could pollute the CPU cache and slow down cache-optimized workloads (e.g., loop tiling [8]). However, while cache-optimization is commonplace in server workloads, it is uncommon on smartphones, especially in the interactive UI-thread. Nonetheless, we evaluate this effect in §8.

2 Related Work

Record and replay (R&R) tools [10,15,16,29,30,35,36,38,50,57] work by recording a user's input and all non-deterministic events (e.g., scheduling), so that the execution can be faithfully replayed. R&R tools do not meet our requirements for a few fundamental reasons. The first is overhead. Among all R&R tools, Reverb [35] reported the best performance, yet its overhead is still 5.5% *on average* (the worst-case is not reported). It works only on JavaScript web applications, where threads communicate using a message-passing interface. When threads share memory, R&R incurs even higher overhead. For instance, DoublePlay [57] reported a worst-case overhead of 11% for network-bound workloads (Apache webserver), 19% for disk-bound workloads (MySQL), and 278% for CPU-bound workloads (SPLASH-2 ocean). To achieve low overhead, some tools [33, 38, 45] do not record all non-determinism which prevents accurate replay. Second, since intermittent performance bugs may take days to occur, R&R traces will grow untenably large. While checkpointing could allow replay from a partial trace, the checkpointing operation itself is expensive [50]. Compared to a call chart, an R&R trace also imposes much larger privacy concerns. Finally, R&R tools require deep integration with the Android runtime and compiler. For instance, applying DoublePlay's approach to Android would require the runtime to run a parallel execution of the application, checkpoint and compare state between the two processes, and so on. Hence, R&R tools would be difficult to maintain within Android.

An attractive alternative is to use hardware-support, like Intel PT or ARM ETM, to record branch-level traces [12,28,60]. These tools have a *worst-case* runtime overhead of 1–2%. However, there are two challenges on ARM devices. First, the semantic gap on Android's runtime complicates the decoding of the branch-level trace, as it only provides the traces of the runtime's execution instead of the application. Second,

hardware support for tracing is restricted to development platforms (most ARM processors on production Android devices do not support the feature) [4].

Only a limited set of bytecode method tracing tools are available on the Android platform. Android Studio's CPU Profiler can trace every method call, but its overhead is incredibly high (a worst-case of 921× in our evaluation), because instead of embedding the tracing logic into the compiled binary, it jumps into the Android runtime after every method call. Internal tracing utilities within Android mostly leverage Java Agent, JVMTI, or equivalent ART instrumentation interfaces to perform method tracing. These mechanisms are also expensive as they force applications to be interpreted only. Aspect-oriented frameworks such as Tai Chi [53] and Logan [55] are also available to intercept method calls at runtime to execute arbitrary tracing code. However, they either require modifications to the application's source code or root access. The fastest available method tracing utility that we are aware of, Nanoscope [54], primarily targets method tracing inside an x86 Android emulator, costing up to 10× higher memory usage and performance overhead, so it is mostly useful in an application development environment.

Some tools are able to perform in-application tracing with low overhead in production. For instance, Firebase performance monitoring [23] collects various metrics (e.g., startup time) and allows developers to insert additional trace points. AppInsight [41] instruments Windows Phone application binaries to log whenever the runtime calls into and returns from application methods. The instrumentation has sufficient detail to allow a server to reconstruct how a user request was processed across different application threads and what the critical path is. These tools typically trace the entire run of an application, but at a low enough granularity that the trace does not grow untenably large. As a result, they are useful for application developers to locate bottlenecks in their application; but the coarseness of the trace may necessitate additional debugging information to locate the exact root cause, especially if the bug is in the underlying systems which are not traced. Timecard [42] goes beyond tracing by using AppInsight's traces to adjust the server's computation quality (in real-time) to meet an end-to-end response deadline.

There are also a few high-performance logging solutions like NanoLog [58] and Log20 [59] that can provide nanosecond-level logging. Both write data to thread-local ring buffers and NanoLog uses a specialized encoding to save space. NanoLog uses only the existing log statements in the application while Log20 can be used to determine where best to place log statements based on profiling the application's usage pattern.[1] In any case, the generated trace is only as detailed as the developers' instrumentation.

Outside of the Android platform, there are many call profiling tools like gprof [27], Fay [17], ftrace [51], perf [39], DTrace [9], and SystemTap [52]. These tools support various degrees of tracing from periodically sampling the call stack to calling user-defined methods using dynamic instrumentation. However, to capture traces that are detailed enough to diagnose intermittent bugs, these tools incur overhead that prevents them from tracing continuously in production systems. These tools typically require calling a method in their instrumentation, whereas Hubble directly inlines the tracing code into each method.

There are a large number of tools designed to trace each request in a distributed system. Examples include Project5 [1], MagPie [5], X-Trace [20], Dapper [47], ÜberTrace [11], and Pivot Tracing [31], as well as commercial tools like Datadog [13] and New Relic [44]. These tools typically embed trace points in critical system or network events, such as RPCs, and record an ID that is unique to each request.

3 Case Studies

We present two case studies to showcase how Hubble helped in diagnosing real-world intermittent performance problems. The first issue was within AppX, a third-party multipurpose messaging, social media, and mobile payment application with over a billion monthly active users. Occasionally, AppX users experienced intermittent UI freezes (janks) of up to two seconds. Engineers detected this problem by monitoring the traces that Systrace continuously collects—namely, performance alerts, sparse trace points, and metrics sampled at low frequencies. Figure 1 (A) shows the available trace points rendered as a method call chart in the Perfetto trace-visualization tool. For the UI thread, this consists of only a few high-level methods within the Android framework. The only conclusion engineers can infer from this data is that the UI thread was blocked for about two seconds during which it was supposed to prepare the layout and content for rendering.

In contrast, the call chart based on Hubble's trace, shown in Figure 1 (B), accurately captures every method call in both the application and the Android runtime. From the canonical method names displayed in the chart, engineers were able to quickly reconstruct the events that occurred before, during, and after the UI jank. First, the user swiped back on the device's screen within AppX (①). Then, AppX initialized a software keyboard to respond to the user's action (②). However, to display the keyboard, the scrollable chat component must be resized (③), and this became the bottleneck. Drilling down further, we can observe the series of method calls responsible for generating the list of on-screen content (④). Specifically, we can see that the UI thread is primarily blocked by various long-running methods belonging to AppX. Now with concrete evidence, our engineers concluded that the root cause was within AppX, and initiated a meaningful collaboration with AppX's developers.

The second issue was a longstanding performance bug

[1] In fact, the initial goal of this project was to integrate Log20 into Huawei's Android platforms.

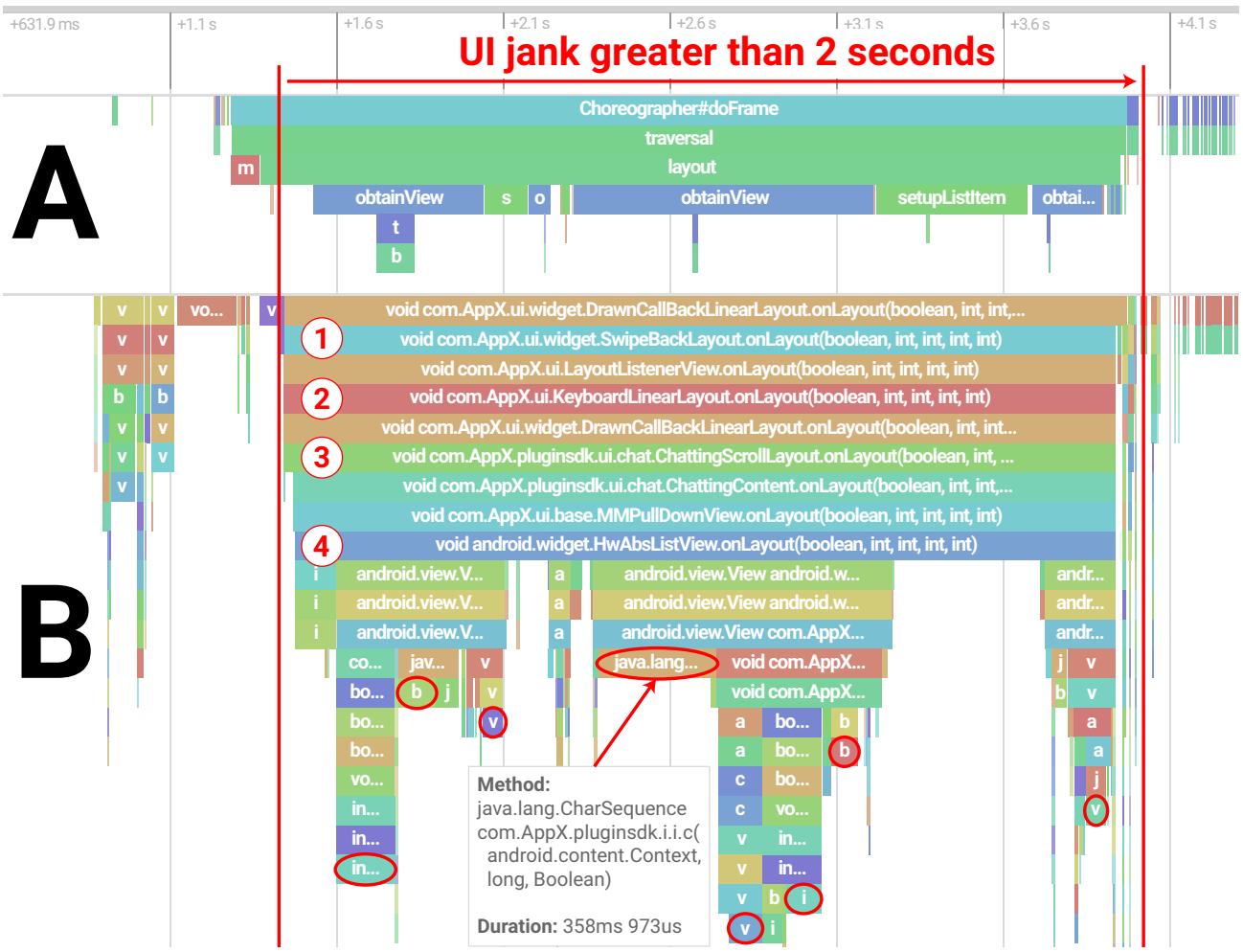


Figure 1: Screenshot of method call charts in Perfetto for the UI thread, which performs all UI and Android framework operations. (A) Traces generated by Systrace, (B) Traces with Hubble. Circled in red are 3rd-party application methods with long execution time. (A) includes *all* of Systrace's trace points recorded during this time period, whereas (B) is filtered to render only approximately 10% of all available methods.

within an internal business teleconferencing application. After the end of a teleconference, the application occasionally froze for up to a second on a small number of user devices. This annoyed users but it was not until months later that a particularly vocal employee reported the issue to management, who then opened a support ticket requesting that the issue be resolved. Our device support engineers attempted to reproduce the problem on their own, but all attempts were unsuccessful. The only method call captured by Systrace was `binder_transaction()`, which does not explain why the issue occurred. Further efforts to collaborate with the disgruntled users were also ineffective as most users were either too busy or otherwise unable to provide more detailed reproduction instructions. A few users were even invited to collaborate with an engineer to reproduce the problem, but the intermittent issue could not be reproduced after multiple attempts.

Several months later, Hubble, in pre-beta at the time, was available for internal use. The disgruntled employees happily consented to deploying Hubble onto their mobile device via an over-the-air Android OS update. Within a few days, performance anomalies were detected and their associated trace data was automatically collected. After a quick glance at Hubble's call chart, the support engineers identified that the teleconferencing software was calling `Thread.sleep()` from the UI thread after sending an Android Binder (IPC) system service call. Closer inspection revealed that immediately after a conference call ended, a series of method calls related to the Audio Manager were performed, prior to the `Thread.sleep()`. This behavior was unexpected and if not for the complete method call trace, which contained both the application and Android framework layers, we would still be stuck with many of our initial theories; e.g., the application could be collecting and sending meeting summary data back to the teleconference service or an unexplained scheduling issue.

With this new information, we brought in a developer with expertise in the Android audio stack. After examining Hub-

ble's call chart, the developer immediately identified the root cause. The problem can only be reproduced under very specific conditions where users must be connected to Bluetooth headsets using a special mode prior to ending the meeting. After the meeting ended, the application immediately rerouted audio to Bluetooth devices connected over the A2DP streaming protocol. This rerouting process requires re-initialization of Bluetooth's SCO (synchronous connection-oriented) link where the `Thread.sleep()` was invoked to wait for the link to be established. We were unfamiliar with these details, but with the help of a developer with the necessary domain knowledge, the issue was promptly fixed by moving the connection and rerouting logic into an asynchronous event handler.

4 Background and Overview

This section first discusses Hubble's design goals and the role it plays in the failure diagnosis process, which helps to understand Hubble's design. We then provide an overview of Hubble, leaving the details to the subsequent sections.

4.1 Goals and Requirements

Hubble's performance overhead and resource usage must be *undetectable* in all real-world usage scenarios. In practice, this translates to two requirements: Hubble's worst-case overhead in real-world scenarios, in terms of both latency and memory usage, should be less than 3%. This target was set by our quality assurance team since they cannot reliably measure overhead below 2–3% on mobile platforms, even under ideal conditions. Nonetheless, this is similar to the target set by other practitioners; Google, for example, reported a 2% overhead budget to deploy tracing tools in production server workloads [32, 48]. The second requirement is that the overhead budget should be respected regardless of whether Hubble is tracing workloads on big or little cores. Besides not being as fast as big cores, little cores also tend to lack advanced features like out-of-order execution. Thus, they enforce stricter restrictions on the tolerable overhead for Hubble. In any case, satisfying the target overhead only allows a tool to pass the deployment planning review. To be deployed in production, the tool needs to go through a systematic procedure consisting of three phases:

1. **Internal testing**. We simulate users using our devices by sending a stream of pseudo-random inputs to a large fleet of physical devices. Each device collects various metrics like application startup times, the number of dropped frames, and so on. Each metric forms a statistical distribution over a large number of trials. We compare the distribution before Hubble was added with the one after. If the differences are statistically insignificant, Hubble has not caused a noticeable change, and we move to phase 2.

2. **Internal beta release**. We push engineering builds to a small group of internal beta testers on their daily-use devices. We ask these users to report any performance regressions they notice and any new performance issues will need to be resolved before moving to the next phase.

3. **Public beta release**. A build is pushed to all beta users (tens of thousands of users), and we monitor all new performance anomaly reports. We only consider Hubble's overhead as undetectable when the beta build does not show a statistically significant increase in reports. Only then can the tool be further released to the entire public.

Android applications are typically distributed as bytecode compiled from high-level languages like Java. Once downloaded, this bytecode is either ahead-of-time (AOT) compiled, or executed within the Android runtime (similar to the Java Virtual Machine). The Android runtime compiles frequently executed code Just-in-Time (JIT), using the same AOT compiler. An already-compiled application could also be re-compiled, if runtime profiling reveals new optimization opportunities. Applications could also contain native libraries, i.e., code that was already compiled into native instructions. Thus, Hubble must be able to operate with only access to the downloaded or generated bytecode or native instructions.

Easy maintainability across Android versions is required. Android is typically updated every six to twelve months, with each new release potentially breaking features or making large-scale changes internally. Thus, Hubble should be modularized and decoupled from the upstream source.

Finally, as the case studies highlight, Hubble needs to be able to trace both the executions of the application and the Android framework to be useful. Ideally, device vendors would only be responsible for analyzing and debugging bugs within Android, and application developers would only be responsible for bugs within the application. The reality, however, is that bugs in the Android framework may manifest themselves in the application and vice versa. Furthermore, system traces are not available to application developers (in order to maintain users' security) and in-application traces may not be available to or easily understandable by device vendors. Exacerbating the issue, application developers and even internal developers at Huawei are reluctant to investigate bug reports without clear evidence that the bug is in their code. Whole-system method traces allow engineers to infer roughly what the application and framework are doing, together, so that the problem scope can be narrowed down to specific call chains and system services. Essentially, Hubble needs to bridge the gap between system and application developers, which in turn, will significantly ease triaging and debugging for both parties.

Overall, these requirements highlight the practical challenges of designing and deploying tracing tools onto a complex user-device platform such as Android.

4.2 The Failure Diagnosis Process

To understand Hubble's utility, we first need to overview the failure diagnosis process. Android devices ship with a set of anomaly detectors to detect common issues like lags in the UI. When an anomaly detector fires, the system saves several pieces of data such as logs, metrics, and traces. At an appropriate time, these data will be uploaded to the device vendor for analysis.

4.2.1 Anomaly Detection

Since Hubble's utility depends on an anomaly detector firing, we first provide background on the detectors available in the Android Open Source Project (AOSP) and our version of Android. There are two branches of anomaly detection mechanisms that device vendors can use in the production environment: Those implemented by Android itself and those implemented by in-house engineering teams. Both branches use information gathered either from the Android runtime layer or from the Linux kernel. In addition, both branches are generally tuned to be conservative to reduce the number of false positives. However, if a severe performance issue occurs, a signal will most likely be raised.

The anomaly detectors implemented in the AOSP have been continuously developed for over a decade. For example, the most frequently used anomaly detector is the UI jank (lag) detector, which has an extremely close correlation to user-observable performance issues. It will alert if a number of consecutive display frames are delayed longer than a predefined threshold. Android officially groups all its tracing, profiling, and anomaly detectors under one umbrella term known as systrace. In production environments, most of these anomaly detection signals and alerts are continuously captured and analyzed in real time.

Internally, we utilize a number of additional black-box anomaly detectors which monitor for a number of kernel level indicators and hardware events. For example, we implemented a system-level, HCI-based detector: Studies show users start to perceive a delay after 400-600ms. So by instrumenting the runtime where (1) a touch is detected by the screen, (2) when the signal is delivered to the application, and (3) the application generates a response, we can accurately measure the delay between (1) and (3) and fire an alert when the delay is longer than 400ms. Furthermore, we can attribute the delay to either signal delivery in the runtime or within the application.

Other black-box anomaly detectors could be as simple as monitoring whether the device has entered the thermal throttling mode. Most detectors, however, don't rely on a single metric. Instead, they correlate multiple metrics. For example, if a detector detects that the current GPU memory bandwidth utilization is high, it then checks other metrics such as the rendering queue backlog length; only if multiple of them suggest an anomaly does the detector fire a warning. Experimental anomaly detectors may further leverage real-time machine-learning monitoring Android runtime metrics like the number of locks held, memory allocation and garbage collection frequency, and so on.

4.2.2 The Utility of Hubble

When Hubble's traces are collected, they are integrated into systrace and Perfetto when presented to engineers with other runtime data. Perfetto and systrace are powerful debugging tools that can visualize a variety of runtime data, including visualizing the method trace as a call chart or flame graph. The tools also have search and analytics (e.g., using SQL) capabilities that allow developers to correlate data from different sources. For instance, developers can cross-examine traces with logs and hardware metrics. Developers can also alert based on traces. For example, one use case of Hubble is to search for the call stack that matches a specific method invocation order, get an average runtime, and alert when it exceeds a threshold. As a result, Hubble is not a standalone tool, nor the only debugging tool. Instead, developers usually start debugging by first examining the data from existing logs and metrics, and some bugs can be resolved with these alone. However, the remaining bugs—typically hard-to-diagnose, intermittent issues—require more insight, which is where Hubble excels.[2]

Key to Hubble's success is the visibility it provides into application and framework-level behaviour, without which engineers cannot triage issues. Hubble's detailed method traces also allow developers to better understand how a bug can be reproduced; with a reproduction, developers can repeatedly reproduce the bug in a development environment (with heavyweight tracing) until the issue is understood.

Nonetheless, there are some limitations to Hubble's utility. We have found Hubble's traces are not as useful in the following cases: (1) if the bug is in the system's native code (which is not traced), (2) if the method-level trace is not fine-grained enough (e.g., an infinite loop without making any function calls), or (3) if a bug is caused by incorrect data-flow (i.e., an incorrect variable value) that does not affect the call path (otherwise it could be inferred by Hubble's trace). However, Hubble's traces can still help developers to significantly narrow down the problem scope (e.g., they can locate the method that contains the infinite loop). In theory, if the distance between the root cause and the symptom is too long, Hubble could miss the cause due to the ring buffer size. However, we have not yet encountered such a case in practice.

[2] We do not have an exact number of issues exclusively resolved by Hubble, because Hubble's traces are integrated into existing debugging tools with other traces. However, we noticed the number of bug tickets containing intermittent and difficult to reproduce bugs quickly dropped after Hubble was first made available.

4.3 Overview of Hubble

Hubble modifies the compiler and interpreter to instrument tracing logic at the entry and exit of every non-inlined bytecode method, whether it is interpreted, ahead-of-time compiled, JIT compiled, or recompiled. Portions of the Android framework itself and factory installed apps, i.e., the apps that are packaged by the OEM vendor, could be already in compiled form instead of bytecode; for these cases, the trace points are embedded at the vendor's site. Hubble can also trace calls made using the JNI (when applications calls into the native libraries and the returns). However, function calls made within native libraries cannot be traced by Hubble.

Hubble adds one system thread, the trace control thread, to each application's process that can turn tracing on or off for any thread in the same process. Although Hubble instruments all bytecode methods, by default, the control thread only turns on tracing for the UI thread, which performs all UI and Android framework operations. At every method entry and exit, Hubble's tracing code writes an entry to a fixed size in-memory ring buffer. When the buffer is full, the buffer pointer will wrap around so the oldest data will be overwritten.

When a performance anomaly detector detects a performance problem, the control thread will be notified. It then notifies the UI thread to stop tracing, preventing useful debugging data prior to the problem from being overwritten. Once tracing has stopped, the control thread flushes the ring buffer to disk, before restarting tracing. The saved trace file could be sent back to Huawei to aid postmortem debugging, or post-processed and analyzed on the device, off the critical path, if a summary needs to be sent.

Each traced thread writes to a private ring buffer local to itself. Hubble keeps at most N buffers in the system, from the N threads that most recently executed in the foreground. Older buffers will be reclaimed by the system. N is configurable and the method trace logic can be programmatically enabled and disabled for individual threads, either via the runtime or by the user application itself. This means that any background threads from almost any process, even short lived ones, can be traced. However, if there are too many concurrent threads being traced, Hubble will run into memory usage issues. To solve this, we could have a ring buffer per core rather than per thread; to differentiate trace points from different threads, we could record the thread's ID (available from a register in the runtime) in each trace point. By default, N is set to 2. This is sufficient to capture both the current foreground and most recent background application's UI threads.

5 In-memory Tracing

This section describes the design and implementation of Hubble's tracing logic. We first explain the information recorded in each trace point and its encoding. We then discuss how we integrated the tracing code into Android's optimizing compiler so that compiler optimizations do not affect our instrumentation.

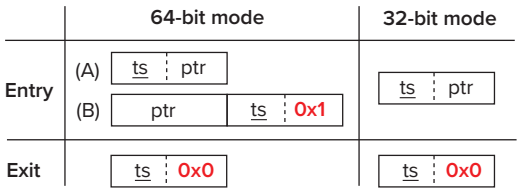


Figure 2: **Format and encoding of trace points** at method entry and exit, and in 64-bit and 32-bit execution modes. "`ts`" and "`ptr`" are timestamp (generic timer count) and method pointer. A solid bordered box represents a 64-bit slot. Underscores represent lossy encodings of timestamps.

5.1 Data Format and Encoding

Figure 2 shows the format of each trace point. As shown, method entry points have a varying encoding depending on the CPU's execution mode and other factors explained later. The CPU will change mode when executing a 32-bit or 64-bit application. Method entry trace points contain a timestamp and a method pointer, while exit points contain a timestamp and the constant `0x0`.

For timestamps, Hubble uses the Generic Timer [3] count instead of the standard system clock. A Generic Timer is a high resolution clock (nanosecond precision) and its tick value can be directly read from a register on modern ARM SoCs. It ticks at a constant frequency regardless of the CPU operation speed and the counter value starts at 0 when reset. When the trace is persisted, Hubble records the current time, which can be used to reconstruct the absolute timestamp of each trace point from the Generic Timer count.

The method pointer is the memory address of a metadata object, ArtMethod, that describes each loaded class-method and can be used to decode a method's canonical name. As part of the ClassLoader initialization process in Android's runtime (ART), an array of ArtMethods is allocated in a memory region outside the managed heap (ignored by garbage collection). ArtMethods can only be added to this array and never be modified nor removed. ART ensures that immediately after entering a method, the address of its ArtMethod is stored in register r0. Since the lifecycle of the main ClassLoader, which is responsible for loading all of the executed bytecode methods, spans the entire duration of the application, we can safely store the ArtMethod pointer in the trace buffer and reconstruct the method name after the trace data is persisted, so long as this happens before the application exits. Note that applications could use additional custom ClassLoaders with shorter lifecycles. If we persist the trace data *after* the custom ClassLoader exits, we could dereference pointers that are no longer valid. To avoid this, we install a cleanup hook for custom ClassLoaders to invalidate the trace buffer (or optionally persist the trace data).

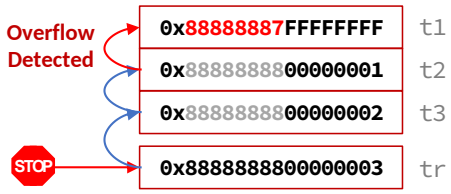


Figure 3: Iteratively Recover Truncated Timestamps.

For each thread that is traced, the control thread allocates storage for the trace in the traced thread's local storage (Java ThreadLocal [37]). This includes a ring buffer and metadata such as where content in the buffer begins and ends. The ring buffer is carved into an array of 64-bit wide integers in both 64-bit and 32-bit mode.

For the timestamp in each trace point, Hubble only stores the lower 32 bits of the Generic Timer counter, regardless of execution mode. (Even in 32-bit mode, the Generic Timer counter is 64 bits wide because the value is fetched from a co-processor that is not subject to the mode change.) Thus, the recorded timestamp may wrap around, which we handle during decoding.

Figure 3 shows how Hubble reconstructs the accurate timestamp from truncated ones. The last timestamp is a reference timestamp (`tr`), which is the complete 64-bit Generic Timer counter value recorded when the trace is persisted. Using `tr`, we can iteratively reconstruct the upper 32 bits of the previous three timestamps: if a previous timestamp has a lower value than the current one (e.g., `t3` versus `tr`), we assume it has the same upper 32 bits; if it has a higher value (e.g., `t1` versus `t2`), we assume a wrap around occurred and the upper 32 bits should be decremented by one.

Theoretically, this could lead to an error: if between two consecutive trace points more than 2^{32} ticks occur, the reconstructed timestamp will be inaccurate. However, this is unlikely to happen in reality. It takes 223.7 seconds on a Qualcomm ARM SoC and a little over 37 minutes on a Huawei-designed SoC for the lower 32-bit Generic Timer counter to tick 2^{32} times. So only if a method executes for more than 223.7 seconds, without calling another method or returning, will an inaccuracy occur.

5.1.1 Format under 64-bit Mode

Hubble uses a variable-width encoding for the ArtMethod pointer when executing in 64-bit mode. In this mode, the pointer is 64 bits; but for real-world applications, the vast majority of the pointers' upper 32 bits have the value `0x0`. We exploited this observation to increase encoding efficiency. When the upper 32 bits are `0x0`, Hubble only records the lower 32 bits of the pointer (Figure 2 (A)). Together with the lower 32 bits of the timer count, a method entry trace point occupies a single 64-bit buffer slot. If the upper 32 bits of the method pointer are not `0x0`, a method entry trace point occupies two buffer slots (Figure 2 (B)). The first 64-bit slot is used to save the complete 64-bit method pointer; in the second slot, the upper 32 bits store the timer count and the lower 32 bits store the constant `0x1`.

The method exit trace point occupies a single 64-bit slot. The upper 32 bits store the timer count, and the lower 32-bit stores `0x0`, indicating it is a method exit trace point.

Traces in this format can always be **unambiguously decoded** in reverse. To decode each trace point, Hubble first checks the lower 32 bits of the previous slot. Depending on whether its value is `0x0`, `0x1`, or another value, Hubble knows that this trace point is either a method exit, a method entry that is two slots wide (Figure 2 (B)), or a method entry that is one slot wide (Figure 2 (A)). `0x0` and `0x1` cannot be method pointers since they are invalid method pointer memory addresses. A method exit point is matched with the corresponding method entry point in a LIFO manner (implemented using a stack). Note that the decoding occurs server-side, after the persisted trace has been sent back.

5.1.2 Format under 32-bit Mode

In 32-bit mode, both method entry and exit trace points use a single buffer slot. The upper 32 bits are always the lower 32 bits of the timer count, like in 64-bit mode. For method entry points, the lower 32 bits store the method pointer, and for method exit points, the lower 32 bits store `0x0`.

5.1.3 Efficient Recording

The tracing logic can be efficiently implemented by a few assembly instructions. For example, Hubble uses only two assembly instructions to store the method entry trace point under 32-bit execution mode:

```
1 MRRC(al, scratch1, scratch0, 0b0001, 0b1111, 0b1110);
2 STRD(r0, scratch1, MemOperand(buffer, 8, PostIndex));
```

The first `MRRC` instruction is used to fetch the 64-bit Generic Timer counter value into two 32-bit CPU registers: `scratch1` and `scratch0` (readers can ignore the other operands). Then a `STRD` instruction is used to (1) store `scratch1`, which contains the lower 32-bits of the Generic Timer counter, and `r0`, which contains the ArtMethod pointer, to the memory address stored in `buffer` register, and (2) increment `buffer` by 8 bytes after the memory operation completes. So after this store instruction, `buffer` will point to the next buffer slot.

Hubble's tracing assembly is directly inlined in the basic block at each method entry and exit. Comparatively, in other profilers that use compiler instrumentations, the instrumented code will call a special tracing function. For example, `gcc -pg` instruments a call to the special function `mcount()`, which is required for tools like `gprof`. While easier to maintain and more portable, the added function call introduces overhead.

When tracing is stopped, the valid portion of the ring buffer is flushed to disk using an *fwrite* call. Three metadata files are generated. First, a complete 64-bit Generic Timer counter value (i.e., the reference timestamp) and the absolute system timestamp are collected at the same time; this facilitates the reconstruction of the actual, non-relative timestamp of each trace point if needed. Then the current buffer position and size are recorded. Finally, Hubble computes a symbol table, mapping each unique ArtMethod pointer value to the method's canonical name.

5.1.4 Alignment

Each trace point is always eight-byte (a word on 64-bit devices) aligned. Eight-byte aligned memory accesses are crucial to achieving the highest performance in both 32-bit and 64-bit mode on modern ARM SoCs. Unaligned accesses take at least one more cycle than a properly aligned memory access. In the worst case, a single unaligned access can cross a cache-line boundary and generate two cache misses or even two consecutive page faults. Worse yet, unaligned memory accesses are an unsupported operation on low-power or older ARM processors, so additional memory accesses and massaging logic are required. Accordingly, we use 32 bits to represent the constants `0x0` and `0x1`, since the performance gains of aligned accesses outweigh encoding inefficiency.

5.2 Hand-optimized Assembly

There are a few reasons to write the tracing logic in assembly. First, it decouples Hubble from the Android compiler's compilation flow. If written in C++, the compiler could move, reorder, or even remove the tracing logic (e.g., the tracing logic accesses global variables without a memory barrier (§6), which is an undefined behavior). By writing the logic in assembly, we can insert it after the compilation stage, bypassing any optimizations that are at odds with the tracing. To do so, early in the compilation stage, instead of generating the actual tracing code, we simply insert a special placeholder instruction at every method entry and exit (including exits due to exceptions); we then configure the Android compiler to exempt this instruction from its later optimization stages. After all the optimizations are performed, we replace this placeholder instruction with the actual tracing instructions. This also makes Hubble easy to maintain, as it is decoupled from any compiler changes that are not backward compatible.

Using assembly also allows us to optimize for both big and little cores. The Android compiler's optimization is heavily biased toward the big core. For example, the compiler skips the architecture-specific optimizations when they are unnecessary on big cores that support out-of-order execution. However, the little cores do not support out-of-order execution, so running the compiled code will result in poor performance. For instance, each trace point needs to check if we are at the end

```
// Initialization
start=0x0;  stop=0x0;

Control thread:                 Trace point:
1  start=buffer;                 7  if (start)
2  wait(signal);                 8    trace...
3  stop = 0x1;                   9    if (stop) {
4  while(stop==0x1)             10      stop=buffer;
5    sleep(..);                 11      start=0x0;
6  persist(..);                 12    }
```

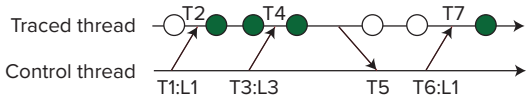


Figure 4: Lock-free Synchronization Protocol.

of the ring buffer (and if so, we need to wrap around). This check requires fetching the value of the ring buffer pointer from memory. If we manually prefetch this pointer (in assembly), it results in an approximate speedup of 35% on the little core. The compiler, however, did not perform this prefetching, because it expects the big core will perform the prefetching automatically.

Finally, because we have domain knowledge of the tracing logic and processor microarchitecture, we can perform better optimizations than the compiler, regardless of whether it is on the big or little core.

6 Tracing Control

Recall a system thread is responsible for notifying the traced thread to turn tracing on or off. The traced thread (e.g., the UI thread) is only responsible for (1) checking whether tracing is turned on, and if so, (2) writing the trace points into the trace buffer, and (3) turning tracing off if necessary. The rest of this section describes how the two threads communicate efficiently without synchronization primitives.

Figure 4 shows the communication between the control thread and the traced thread. Lines 1–6 are the control thread's logic, whereas lines 7–12 are executed at every trace point in the traced thread. Hubble uses two eventually-consistent, shared variables, `start` and `stop`. `start` is unidirectional, i.e., it is set by the control thread and read by the traced thread, and `stop` is bidirectional, as it can be set and read by both threads. Initially, both variables are set to `0x0`. To start tracing, the control thread sets `start` to the *address of the next buffer slot* (line 1 in Figure 4), and waits for a signal to stop tracing. Therefore, the value of `start` indicates two things: whether tracing is on or off, or the buffer position. At each trace point, the traced thread first checks if `start` is `0x0`, and only proceeds with tracing if it is not (line 7).

To turn tracing off, the control thread sets `stop` to `0x1` (line 3 in Figure 4), and then enters a polling loop until `stop` is changed to a value *greater than* `0x1` (lines 4–5). In the meantime, the traced thread performs tracing and evaluates

the value of `stop` at the end of every trace point (line 9). Once the traced thread detects that `stop` was changed to a non-zero value, it enters the logic to stop tracing. The traced thread first sets `stop` to the address of the current buffer pointer, i.e., the end position of the buffer, at line 10. So `stop` also serves dual purposes: whether tracing should stop (with value 0 or 1), or the buffer end position. (Note that `0x0` and `0x1` are invalid buffer memory addresses, so after line 10, `stop` will be greater than `0x1`.) Then the traced thread sets `start` to `0x0` at line 11, to guarantee that tracing will be disabled immediately. Finally, the control thread detects that the traced thread has stopped tracing, so it can persist the trace or clean up the ring buffer.

Figure 4 also shows an example trace control-flow. Each circle represents a trace point, with filled and blank shading indicating whether trace data is written or not. At the beginning, tracing is off. At T1, the control thread turns tracing on at line 1 (L1) by setting `start` to a non-zero value. This new value is propagated to the traced thread at time T2, as the result of eventual consistency in the memory cache coherence protocol. Then, the following three trace points are written to buffer. At T3, the control thread turns tracing off by setting `stop` to `0x1`, which is propagated to the traced thread at T4. The traced thread then executes lines 10–11, and at T5, the control thread detects that `stop` was changed to a value greater than `0x1`; so it breaks out of the polling loop and persists the trace. After the trace is persisted, the control thread restarts tracing at time T6 (line 1).

This design is highly efficient. Each trace point needs to check the values of `start` and `stop` only if the trace has been started. `start` and `stop` are regular shared variables that are almost always cached. In comparison, any alternative design that uses synchronization primitives or atomic variables would introduce much higher overhead in each trace point, which is on the critical path.

Since tracing is stopped and the current ring buffer location is written to the `stop` variable by the traced thread itself, no additional trace point will be written to the buffer afterwards and the buffer metadata will be consistent. For example, if the last trace point is a 64-bit method entry occupying two slots, it is guaranteed that both slots are written with the buffer pointer correctly incremented before tracing is stopped.

If the traced thread is executing native code, either through the JNI or a custom ClassLoader, it cannot respond to the control thread's stop tracing request, because the logic to stop tracing is only instrumented in bytecode methods. Therefore, the control thread further checks whether the traced thread is in native execution when it attempts to stop tracing. If so, the control thread will first obtain ART's state transition lock that prevents the traced thread's execution from changing state, i.e., from native execution back to the bytecode world (either the interpreter or compiled code). Then the control thread forcibly copies the buffer position to `stop`, and sets `start` to `0x0`, followed by a memory fence. Finally, the control thread can release the state transition lock. A subtle data race could occur during state transition where just before the lock is obtained, the traced thread transitions back to the bytecode world. Debugging this unfortunately took weeks, but we fixed it by rechecking the traced thread's execution state after obtaining the lock.

7 Privacy and Security

Security and privacy are some of our top priorities. Hubble does not collect personally identifying information, such as phone numbers or user IDs. Hubble's traces only contain method names and timestamps, there are no actual *data values*, not even parameter values. Widely-adopted error reporting systems like Windows Error Reporting (WER) [21], MacOS' crash report [2], or the Mozilla Crash Reporter [34], record a subset of the memory state or often collect system logs. In comparison, Hubble's traces are far less sensitive. Similar to WER and other widely-adopted error reporting systems, Hubble uses an informed consent policy.

Even when user consent is given, Hubble further strives to minimize the amount of data that leaves the device. Hubble has the capability to perform the same analyses that are performed server-side, locally on a user's device, with only a summary being sent back to the vendor. For example, Hubble can quickly scan the trace files and compute the top methods with the longest "self-execution-time", or it can automatically isolate and extract the longest method call chains from when a performance anomaly occurred. Performance bug models could be distributed to client devices, containing "signatures" of problematic method names or method call chains, and if there is a match, statistics could be sent back instead of the complete trace.

Hubble also exploits many built-in data security features in Android and the Linux kernel to protect trace data. The traces are stored inside an application-private storage area that is protected by the kernel-level application sandbox. Only the application itself with matching its UID, device vendors, and application developers—when they configure their mobile device in debug mode—have access to the trace files.

8 Evaluation

Hubble has been repeatedly tested on Huawei's performance testing framework, which included the top 100 popular applications, with workloads including startup, stress testing (simulated random screen touches at a high rate), and normal usage simulations, on all supported devices. Overall, we have found Hubble's overhead is statistically insignificant in real-world use-cases. Hubble tracing is now enabled by default in all Huawei testing frameworks.

We have designed a few experiments to stress test and study Hubble's runtime characteristics, aiming to answer four questions: (1) What is the runtime cost of Hubble's tracing?

(2) What is Hubble's effect on cache behavior and memory bandwidth? (3) What is Hubble's overhead in the most demanding real-world scenarios? (4) How long of an execution trace can be stored in the ring buffer? We did not evaluate power consumption. Despite best-effort attempts, we could not reliably observe battery overhead in any experiments. Huawei's devices are shipped with aggressively tuned power-saving profiles and thus far, we have not observed an increase in reports of battery drain.

Unless otherwise specified, experiments were performed on a Google Pixel 1 phone that is well-supported by the open-source version of Android (AOSP). The phone contains a Qualcomm Snapdragon 821 processor with two high-performance cores each with a 64 KB L1 (divided equally for instructions and data) and 1.5 MB L2 cache, and two low-power cores each with a 64 KB L1 and 512 KB L2 cache.

We compared three execution modes: (1) baseline – the phone running *unmodified* Android; (2) tracing off – Hubble is enabled and applications are instrumented, but tracing is turned off; and (3) tracing on. Baseline experiments were performed on AOSP's android-10.0.0_r2 [56] branch. We recompiled the same branch with Hubble enabled.

Hubble's overhead could only be measured reliably in CPU-intensive and unrealistic microbenchmarks. Repeatedly running the two microbenchmarks in §8.1 and §8.2 causes the CPU to quickly reduce its clock speed due to severe thermal throttling. To improve the validity and reproducibility of the experiments, we placed the phone on bags of ice water.

8.1 Trace Point Overhead

Hubble's tracing overhead is amortized by the amount of work performed by the traced method. Since Hubble's tracing logic does not impose any dependencies on the traced method, nor does it use synchronization primitives on the critical path, the amortization effect will be enlarged by the deeper CPU pipeline. We evaluated both the cost of an individual trace point as well as the overall runtime overhead as the method performs more work. For comparison, we also evaluated Android's built-in method tracing utility, typically invoked via Android Studio's CPU profiler, henceforth referred to ASMT.

Listing 1 shows the method used. The amount of work done can be controlled through the `work` parameter. To prevent the method from being inlined by the JIT compiler, we added tail-recursion on line 5. In addition, we executed the method with a `depth` of `10` since the compiler still performs inlining at lower depths. `sum` is carried across calls to ensure that the loop is not optimized away by dead code elimination.

We ran the method with `work` values of 0, 1, 10, 100, and 1,000. We measured the runtime of two billion iterations. The cost of a trace point is calculated as the overhead of the 0-work experiment divided by two, since each method call contains a method-entry and method-exit trace point. To ensure the method is compiled by the JIT compiler before evaluation, we ran the experiment until its runtime stabilized to a maximum variance of five percent. The method is then executed ten times for each experiment.

		Average Cost (ns)	Standard Deviation (ns)	Performance Overhead (%)
ASMT	**32-bit**	3,911.575	59.2450	920,587%
Tracing ON	**64-bit**	3,366.050	57.8026	748,510%
Hubble Method	**32-bit**	0.725	0.0551	171%
Tracing ON	**64-bit**	0.650	0.0023	145%
Hubble Method	**32-bit**	0.001	0.0030	0%
Tracing OFF	**64-bit**	0.008	0.0027	2%

Table 2: Cost of a Single Trace Point

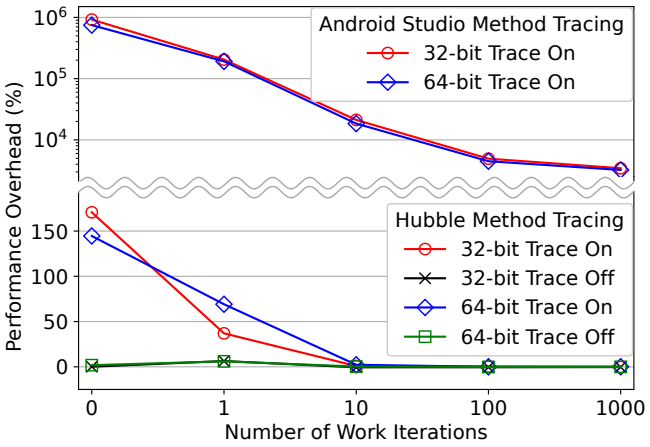


Figure 5: Performance Overhead Over Work Iterations

```
1 public long Test(int depth, int work, long sum) {
2   for (int i = 0; i < work; i++) {
3     sum *= i - 1; sum /= i + 2;
4   }
5   if (depth > 1) return Test(depth - 1, work, sum);
6   return sum;
7 }
```

Listing 1: Program used for measurement.

Table 2 shows the results of the 0-work experiment, with the other work values in Figure 5. The 0-work experiment shows that on average, each Hubble trace point costs less than one nanosecond when tracing is on, and less than 10 picoseconds when tracing is off. This is far less than ASMT's overhead which is on the order of microseconds. Figure 5 shows the amortization effect: as the amount of work done by the method is increased, Hubble's tracing overhead percentage decreases quickly. Note that in reality, small methods like this would likely be inlined, excluding them from being traced.

8.2 Cache Effects Microbenchmark

We used matrix-multiplication (MM) to measure Hubble's effects on the cache. MM is a classic workload that can either benefit heavily from caching or suffer ample cache misses [8]. When multiplying large matrices, a naïve implementation

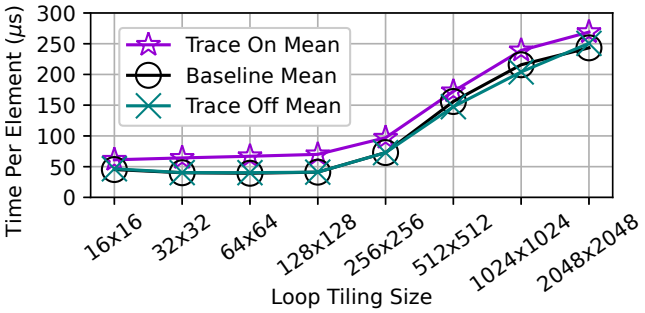


Figure 6: Cache and Memory Effects

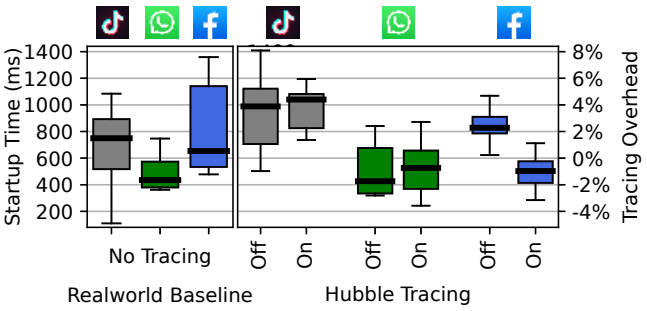


Figure 7: Application Startup Time

causes many unnecessary cache misses. However, the majority of these cache misses can be avoided using loop-tiling, i.e., partition each matrix into many small tiles, where each fits in the cache, and perform all accesses on one tile before moving on to the next. We examined Hubble's effect on each level of the cache by gradually increasing the tile size.

We evaluated Hubble's effect on MM with eight different tile sizes: 16×16, .., 2048×2048. The input matrices are 2048×2048, and each element is a four-byte integer. This means tile sizes 64×64 and below fit within the L1 cache; tile sizes 256×256 and below fit within L2; and all remaining tile sizes exceed both cache levels. To evaluate the highest amount of interleaved memory-contention that Hubble may have with MM, we performed each multiply and add operation inside a method such that two trace points are produced for each step of MM. We also inserted a dummy tail recursion call so that the JIT compiler does not inline the method. For each tile size, we ran the experiment five times. We did not compare with ASMT because it was too slow.

Figure 6 shows the results. With Hubble's tracing turned off, we could not reliably observe any overhead. With Hubble's tracing turned on, for the smallest tile size that fits within the L1 cache, Hubble has a min / max / mean overhead of 41% / 70% / 54%. When the tile size still fits within the L2 cache at 128×128, the overhead increased slightly to a min / max / mean of 64% / 83% / 70%. Finally, when the tile size is much larger than the L2 cache, caching is no longer effective. In this region, the increased execution time when tracing is turned on did not deviate significantly from smaller tile sizes, but the amortized overhead decreased.

Thus, in the absolute worst case scenarios, Hubble indeed affects programs heavily optimized for caching and, to some extent, memory-bound programs. However, in practice, similar small methods invoked in a tight loop would be inlined and excluded from tracing, not to mention that such loop-tiling is unlikely to be used in an application's UI thread.

8.3 Startup Overhead Macrobenchmark

We measured Hubble's overhead on application startup, one of the most demanding but realistic workloads for a method tracing tool since it comprises hundreds of thousands of method calls in a short period of time. These methods perform data loading and processing to prepare the application's UI and are often optimized to ensure the application loads quickly [22].

Since the performance of the application startup process varies significantly in practice, we took additional measures to minimize variation across benchmark runs. Specifically, we ran all experiments while disconnected from the network, eliminating variance introduced by network connections. We launched the target application repeatedly until its startup time stabilized to within a maximum variance of 5% (without these measures, the normal variance can be as much as 100% as shown on the left hand side of Figure 7). Each application was launched programmatically, avoiding any extraneous touch input that would occur with manual interactions. The startup time was obtained from a syslog message that indicates the duration from when the application process launched to the time after the application's UI has been drawn on the screen. To force cold starts (where the application starts completely unloaded), we manually killed each application before starting it again. Furthermore, we performed tests in quick succession to encourage the scheduler to place the application process on the performance-oriented CPU core operating at the maximum clock speed.

We ran the benchmark on the three applications that had the most downloads in 2020 [6]: TikTok, WhatsApp, and Facebook. The results are presented as a box and whisker chart on the right hand size in Figure 7. As the figure shows, the measured startup times vary considerably. To determine if Hubble causes a statistically significant difference in application startup time in our tightly controlled test environment, we performed two single-tailed dependent (paired sample) t-tests with a significance level of 5%. The t-test on the results of tracing turned off produced a p-value of 14.25% and the t-test of tracing turned on produced a p-value of 33.18%, both of which exceed the 5% threshold. Thus, we cannot conclude that Hubble causes a statistically significant difference in application startup time. In contrast, ASMT increased the average startup time of the three applications by approximately 10 times.

Although application startup overhead fluctuates significantly under real world scenarios, the number of methods executed remains nearly constant. When disconnected from

the network, TikTok, WhatsApp, and Facebook filled 6.0 MB, 3.8 MB, 6.4 MB of Hubble's ring buffer respectively; this corresponds to roughly 400,000, 250,000, and 420,000 methods invocations. When connected to the internet, the ring buffer content increased to 14 MB, 5.1 MB, and 11 MB because the applications loaded the user's content. In all three applications, the 32 MB of ring buffer proved to be more than sufficient to capture the entire application startup sequence. In Huawei's Hubble deployment, the 32 MB trace buffer is able to store the duration of almost all application startup and intermittent performance anomalies that our support engineers have encountered.

The results of the macrobenchmark were also in-line with results from our automated performance-regression testing, as well as feedback from support engineers and application developers. Recall that in part one of Huawei's three-phase deployment process (§4.1), we ran automated tests across a large fleet of devices and any significant statistical deviation in the results will prevent a new build from being deployed. In the automated performance-regression tests, we measured the application startup-time (both cold and warm startup) of the 100 most-downloaded third-party applications in addition to all our own applications. We categorized startup times into increments of 500 ms and count the number of applications that fall in each increment. After Hubble's deployment, we have not recorded any statistically-significant changes in the number of applications in each bucket for both cold and warm startup times.

The choice of 500 ms may seem high; however, Farrer *et al.* showed that users do not feel any loss of control (i.e., that an application is not responding to their action) until the response times reach approximately 350 ms [19], and users feel like they have completely lost control when response times exceed approximately 750 ms. Thus, our QA teams (and others [43]) have found that 500 ms increments are a good categorization to qualitatively evaluate loading speed—response times below 500 ms are considered excellent, 500–1000 ms is considered good, and above one second is considered slow.

9 Experiences

Hubble was shipped in the production branch of Huawei's Android system in August, 2020. An early prototype was merged into the main development branch in 2019, and engineers have been using it since. Huawei also runs a beta program where users can receive new features before public release. There are currently tens of thousands of beta users, and Hubble is enabled on their daily-use devices. For other end users, Hubble can only be enabled with their express consent.

The trace collection frequencies and retention policies vary depending on the type of users, the level of consent granted, operating region and local regulations, and device model. Internal beta users may not have any data upload restrictions. However, there are often additional restrictions on public users (including those beta users that are outside of Huawei). A common policy is that each user device can upload at most three traces per week. Which three traces to upload is configurable. For instance, sometimes there is a targeted campaign to improve specific applications, so in that case, only traces of anomalies for those applications are uploaded; other times we collect traces for anomalies whose symptoms are extremely severe; or, in the default case, we collect the first three anomalies detected. Although three traces is a low threshold, with a large user base, we are usually able to collect one or a few traces for each important issue.

Besides debugging production issues, Hubble is equally useful for debugging problems discovered during automated testing. Before Hubble, developers used ASMT to debug performance regressions, but due to its overhead it could only be enabled when debugging. This is cumbersome, and many problems simply could not be reproduced while debugging or worse, new issues would appear with ASMT enabled. Now, whenever a performance regression is detected, Hubble's traces are automatically collected, helping developers quickly narrow down the root cause without reproducing the issue.

A happy accident of implementing the tracing in assembly was that we discovered a bug in ARM's reference design on an older CPU model. While optimizing and testing the tracing assembly on a large number of devices, we found that when a specific permutation of 32-bit assembly instructions is used together with the Generic Timer counter, a segmentation fault could occur on the out-of-order performance cores. The bug was confirmed by the chip design team and fixed in later CPU models. On the buggy CPU model, we work around the issue by using an `ISB` instruction to flush the CPU pipeline after fetching the Generic Timer counter.

10 Concluding Remarks

Call profilers are known to be useful in debugging, however, their use has been limited to the development environment as a result of their overhead. Hubble shows that by leveraging Android's on-device compilation process, a just-in-time flushing strategy, and together with careful system-level design and engineering, we can achieve a highly efficient tool that can collect fine-grained call traces even in production environments. Hubble has proved its usefulness by significantly easing engineers' postmortem debugging processes.

Acknowledgements

We thank our shepherd Jonathan Mace and the anonymous reviewers for their insightful comments. Adrian Chiu provided help for us to understand the internals of a language runtime and its JIT compiler. This research was supported by a contract between Huawei and University of Toronto.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th Symposium on Operating Systems Principles*, SOSP '03, pages 74–89. ACM, October 2003.

[2] Apple Inc. *Diagnosing Issues Using Crash Reports and Device Logs*, May 2021. https://developer.apple.com/documentation/xcode/diagnosing-issues-using-crash-reports-and-device-logs.

[3] Arm Limited. *AArch64 Programmer's Guides: Generic Timer*, August 2019. https://documentation-service.arm.com/static/600eb3264ccc190e5e68023a.

[4] Arm Limited. *Arm® Architecture Reference Manual*, July 2021. https://documentation-service.arm.com/static/611fa684674a052ae36c7c91.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, pages 259–272. USENIX Association, December 2004.

[6] Adam Blacker. Worldwide & US Download Leaders 2020. January 2021. https://blog.apptopia.com/worldwide-us-download-leaders-2020.

[7] Brendan Gregg. *Linux perf Examples: 4.3 JIT Symbols (Java, Node.js)*, July 2020. https://www.brendangregg.com/perf.html#JIT_Symbols.

[8] Randal E. Bryant and David R. O'Hallaron. Computer Systems: A Programmer's Perspective. chapter 6, pages 615–629. Pearson, 2nd edition, 2011.

[9] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 10th USENIX Annual Technical Conference*, USENIX ATC '04, pages 15–28. USENIX Association, June 2004.

[10] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, pages 48–59. ACM, August 1998.

[11] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 217–231. USENIX Association, October 2014.

[12] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 17–32. USENIX Association, October 2018.

[13] Datadog. Cloud Monitoring as a Service. https://www.datadoghq.com/.

[14] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In *Proceedings of the 24th Symposium on Operating Systems Principles*, SOSP '13, pages 33–48. ACM, November 2013.

[15] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 211–224. USENIX Association, December 2002.

[16] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments*, VEE '08, pages 121–130. ACM, March 2008.

[17] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *Proceedings of the 23rd Symposium on Operating Systems Principles*, SOSP '11, pages 311–326. ACM, October 2011.

[18] Facebook, Inc. Profilo - An Android Performance Library. https://facebookincubator.github.io/profilo/.

[19] Chlöé Farrer, G Valentin, and Jean-Michel Hupé. The Time Windows of the Sense of Agency. *Consciousness and Cognition*, 22(4):1431–1441, December 2013.

[20] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, NSDI '07, pages 271–284. USENIX Association, April 2007.

[21] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging

in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116. ACM, October 2009.

[22] Google LLC. *App Startup Time*, April 2021. https://developer.android.com/topic/performance/vitals/launch-time.

[23] Google LLC. *Firebase Performance Monitoring*, April 2021. https://firebase.google.com/docs/perf-mon.

[24] Google LLC. *Inspect CPU activity with CPU Profiler*, May 2021. https://developer.android.com/studio/profile/cpu-profiler.

[25] Google LLC. *Overview of System Tracing*, May 2021. https://developer.android.com/topic/performance/tracing.

[26] Google LLC. *Simpleperf Profiling Tool: JIT Symbols*, September 2021. https://android.googlesource.com/platform/system/extras/+/ec8d549d4c4300dcfb4e12353eccbeba17bf7725/simpleperf/doc/jit_symbols.md.

[27] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126. ACM, June 1982.

[28] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 344–360. ACM, October 2015.

[29] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 463–474. ACM, June 2012.

[30] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient Online Multiprocessor Replayvia Speculation and External Determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 77–90. ACM, March 2010.

[31] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393. ACM, October 2015.

[32] Gabriel Marin, Alexey Alexandrov, and Tipp Moseley. Break Dancing: Low Overhead, Architecture Neutral Software Branch Tracing. In *Proceedings of the 22nd Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '21, pages 122–133. ACM, June 2021.

[33] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 693–708. ACM, April 2017.

[34] Mozilla. Mozilla Crash Reporter, May 2021. https://support.mozilla.org/en-US/kb/mozillacrashreporter.

[35] Ravi Netravali and James Mickens. Reverb: Speculative Debugging for Web Applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 428–440. ACM, November 2019.

[36] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record and Replay for Deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 377–389. USENIX Association, July 2017.

[37] Oracle Corporation. *ThreadLocal (Java Platform SE 7)*, December 2020. https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html.

[38] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192. ACM, October 2009.

[39] *Perf Wiki*, June 2020. https://perf.wiki.kernel.org/index.php/Main_Page.

[40] Perfetto - System profiling, App Tracing and Trace Analysis. https://perfetto.dev/.

[41] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile App Performance Monitoring in the

Wild. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 107–120. USENIX Association, October 2012.

[42] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling User-Perceived Delays in Server-Based Mobile Applications. In *Proceedings of the 24th Symposium on Operating Systems Principles*, SOSP '13, pages 85–100. ACM, November 2013.

[43] Raygun. *Real User Monitoring Performance Metrics*, May 2022. https://raygun.com/documentation/product-guides/real-user-monitoring/for-web/performance-metrics/.

[44] New Relic. New Relic®. https://newrelic.com/.

[45] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[46] Kedar Sadekar. Netflix Engineering Blog: Scalable Logging and Tracking. June 2012. https://netflixtechblog.com/scalable-logging-and-tracking-882bde0ddca2.

[47] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., April 2010.

[48] Richard L. Sites. Datacenter Computers - Modern Challenges in CPU Design. Video, February 2015. https://vimeo.com/121396406.

[49] Joel Spolsky. How Microsoft Lost the API War. June 2004. https://www.joelonsoftware.com/2004/06/13/how-microsoft-lost-the-api-war/.

[50] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the 10th USENIX Annual Technical Conference*, USENIX ATC '04, pages 29–44. USENIX Association, June 2004.

[51] Steven Rostedt. *ftrace - Function Tracer*, July 2017. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[52] SystemTap. https://sourceware.org/systemtap/.

[53] *Tai Chi*, May 2020. https://taichi.cool/doc/.

[54] Leland Takamine and Brian Attwell. Introducing Nanoscope: An Extremely Accurate Method Tracing Tool for Android. April 2018. https://eng.uber.com/nanoscope/.

[55] Jiang Tenglicheng. Logan: Meituan Open Source Mobile Terminal Basic Log Library. October 2018. https://tech.meituan.com/2018/10/11/logan-open-source.html.

[56] The Android Open Source Project. Android 10.0.0 Release 2, 2019. https://android.googlesource.com/platform/build/+/refs/tags/android-10.0.0_r2.

[57] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 15–26. ACM, March 2011.

[58] Stephen Yang, Seo Jin Park, and John Ousterhout. NanoLog: A Nanosecond Scale Logging System. In *2018 USENIX Annual Technical Conference*, USENIX ATC '18, pages 335–350. USENIX Association, July 2018.

[59] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 565–581. ACM, October 2017.

[60] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1155–1170. ACM, June 2021.

Jawa: Web Archival in the Era of JavaScript

Ayush Goel[1], Jingyuan Zhu[1], Ravi Netravali[2], Harsha V. Madhyastha[1]
[1]*University of Michigan*, [2]*Princeton University*

Abstract—**By repeatedly crawling and saving web pages over time, web archives (such as the Internet Archive) enable users to visit historical versions of any page. In this paper, we point out that existing web archives are not well designed to cope with the widespread presence of JavaScript on the web. Some archives store petabytes of JavaScript code, and yet many pages render incorrectly when users load them. Other archives which store the end-state of page loads (e.g., screen captures) break post-load interactions implemented in JavaScript.**

To address these problems, we present Jawa, a new design for web archives which significantly reduces the storage necessary to save modern web pages while also improving the fidelity with which archived pages are served. Key to enabling Jawa's use at scale are our observations on a) the forms of non-determinism which impair the execution of JavaScript on archived pages, and b) the ways in which JavaScript's execution fundamentally differs between live web pages and their archived copies. On a corpus of 1 million archived pages, Jawa reduces overall storage needs by 41%, when compared to the techniques currently used by the Internet Archive.

1 Introduction

URLs are brittle pointers to information on the web. Over time, a page may cease to exist at the URL where it was originally available [44, 62] or the content available at that URL might change due to the page being modified [58, 36].

Therefore, web archives play a key role in the web ecosystem, enabling users to lookup the content that existed at any particular URL at various times in the past. Web archives are used for a wide variety of use cases, such as web-data analytics, genealogical analysis, and even as legal evidence [40]. To support these uses, a number of organizations—cultural heritage institutions, national libraries, and public museums—operate web archives to ensure long-term preservation of content on the web. A recent survey estimates that there are 119 web archives in the United States alone [35].

The largest and most popular of these archives, Internet Archive (IA), has archived over 600 billion web pages to date, storing data in excess of 100 petabytes [13]. It repeatedly crawls web pages over time and saves many snapshots of every page. For every page snapshot, IA first downloads all resources (e.g., HTMLs, CSS stylesheets, JavaScripts, images) on the page). It stores these resources after rewriting all URL references to point to the copy hosted by the archive. When a user wants to later view any stored snapshot of a page, the user's browser loads the snapshot from IA in the same manner as it would load any page on the live web.

In this paper, we argue that this modus operandi no longer suffices due to the preponderance of JavaScript on modern web pages [18, 38, 53]. Specifically, the widespread use of JavaScript hinders web archives from satisfying two of their primary objectives: 1) to capture and save as much of the web as feasible, and 2) to ensure that archived page snapshots faithfully mimic the original page.

- **Higher operational costs:** First, the total number of bytes on the median web page has more than tripled over the last decade [10]. A significant contributor to this increase has been the increased usage of JavaScript. For example, across Internet Archive's copies of the home pages of 300 randomly sampled sites, we see that JavaScript accounts for 44% of the bytes on the median page in 2020, as compared to 20% in 2000 (§2). Since web archives are typically run by non-profit institutions with limited budgets, needing to store more bytes per page reduces the number of pages they can crawl and archive.
- **Poor page fidelity:** The archived copies of many JavaScript-heavy pages render with missing images and improperly laid out content (§2.1). This occurs due to the non-deterministic execution of JavaScript; when a user loads an archived copy of a page, the resource URLs requested by the user's browser can differ from those saved by the archive when it crawled the page. Consequently, the web archive returns errors for some of the requested resources. Due to the complex dependencies between the resources on a page [65, 34, 54], one failed resource fetch often has a cascading effect on the rest of the page load.

The challenge in holistically addressing both problems is that trying to reduce storage overheads by not saving some of the JavaScript found on crawled pages risks further degrading fidelity. A web archive could statically or symbolically analyze the JavaScript code on every page to identify what subset is necessary to preserve correctness in *all* potential loads of the page. However, the computational overheads of such methods [42, 48] render them impractical at the scale of a web archive, e.g., the Internet Archive crawls roughly 5000 pages per second [64]. To jointly address JavaScript's adverse impacts on storage and fidelity using computationally lightweight methods, we observe and leverage three fundamental ways in which JavaScript's execution on archived pages differs from that on the live web.

First, a significant fraction of JavaScript is dedicated to either sending user data to a page's origin servers or processing dynamically constructed server responses, e.g., to enable

users to post comments or to push notifications. Any such functionality cannot work on archived pages, and therefore, the associated code need not be stored by web archives. Fortunately, the JavaScript code on any page is typically partitioned into several files, and we find that most of the code that will be non-functional in the context of a web archive is cleanly compartmentalized into a subset of these files that exhibit identifiable patterns in their URLs. Consequently, we show that web archives can efficiently, and safely, prune unnecessary JavaScripts by relying on URL-based filters to identify and discard JavaScript source files.

Second, many lines of JavaScript code are executed only in certain control flows, e.g., when a page is loaded on a smartphone, and not on a desktop. But, among the various sources of non-determinism that dictate whether or not a specific line might get executed, some sources are absent in loads of archived page snapshots; clients maintain no state across loads and server responses for the same request URL do not vary. Moreover, a web archive should actively eliminate those sources of non-determinism which can cause clients to request different resource URLs than those crawled. Thanks to the resulting reduction in non-determinism, we find that much of the JavaScript code on an archived page will never be exercised in any load of that page, making it moot for a web archive to store such code.

Lastly, a critical use of JavaScript is to enable users to interact with a page after the page's load has completed. On live pages, identifying *all* the code used to support such interactions is generally challenging because the code that is exercised varies based on how users interact with the page. For example, the input given to a search bar determines the server's response; based on the number of search results, JavaScript for paginating the results may or may not get executed. In contrast, we find that the subset of interactions that do work on archived pages (e.g., navigational menus and image carousels) distinctly differ from those that do not with respect to the properties of the page state they access. This greatly simplifies the task of identifying the code necessary to preserve post-load interactions.

Based on our three observations, we design and implement Jawa (JavaScript-aware web archive), a system for crawling and saving web pages. Jawa enables web archives to save many more pages than they could today for the same cost, e.g., it reduces the total amount of storage necessary to store a corpus of 1 million web pages by 41%. Importantly, Jawa enables this reduction both while increasing the rate at which pages can be crawled by 39% and significantly improving the fidelity of archived pages: for the vast majority of archived pages, Jawa ensures that the page is rendered in a manner identical to how it was when the page was crawled, and all page functionality that can possibly work on an archived page does work. Source code for Jawa, including scripts to reproduce the key results in the paper, are available at https://github.com/goelayu/Jawa.

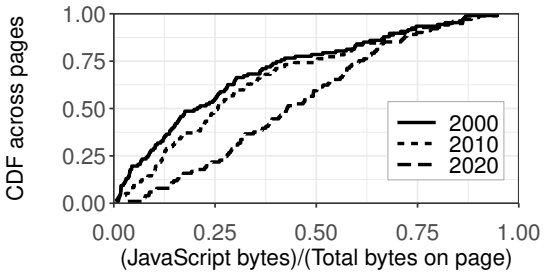


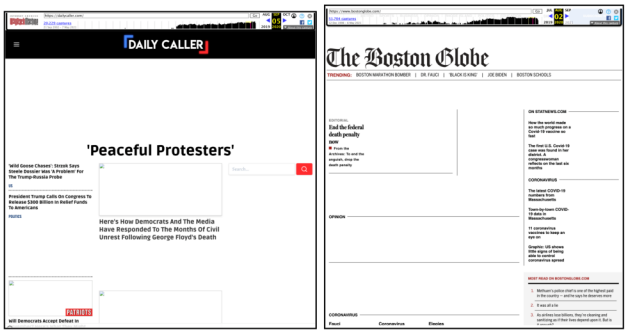
Figure 1: **Across the landing pages of 300 sites, distribution of fraction of bytes on the page accounted for by JavaScript.**

2 Background and Motivation

As mentioned earlier, the Internet Archive (IA) is the largest and most popular web archive in the world today. For every page that it crawls, IA stores all the individual resources on that page (such as HTMLs, CSS stylesheets, JavaScript files, and images) in the Web ARChival format (also known as the WARC format [22]). Client browsers can load archived pages from IA's Wayback Machine [24] in a manner identical to how they do on the live web. When the Wayback Machine receives a request for any resource, it looks up an internal index to locate the WARC record for this resource and then responds along with relevant HTTP headers. IA rewrites all resource files so that all statically embedded URLs point to IA's web servers. For URLs which are dynamically generated via JavaScript, IA rewrites them on the fly using client-side API shims.

This architecture sufficed when IA began operating two decades ago. However, the web today is very different. In particular, JavaScript (JS) has become significantly more common. For example, Figure 1 shows that JS accounts for 44% of the bytes on the median page today; up from 20% in 2000. In this section, we show that this increase in JS hinders the ability of web archives to meet their two primary objectives: 1) to crawl and capture as much of the web as possible, and 2) to preserve page fidelity, i.e., when an archived page is loaded by a user, it should ideally match the page as it was crawled, both in visual (how the page looks) and functional (user interactions supported on the page) aspects.

To support our claims, in this section (and in the rest of the paper), we consider pages from 300 sites, comprising 100 randomly chosen sites from each of three ranges from Alexa's site rankings: [1, 1000], [1000, 100K], and [100K, 1M]. Using these 300 sites, we construct two corpuses. $Corpus_{3K}$ contains one of IA's copies from September 2021 for 1 landing and 9 internal pages per site. $Corpus_{1M}$ contains 3500 page snapshots for each site out of all of IA's page snapshots from September 2020. Note that both corpuses contain a mix of old and new pages. Though both corpuses contain page snapshots which were archived in the last couple of years, many of these pages were created before then. This is because IA recrawls pages over time to track changes to page content.



(a) dailycaller.com [21] (b) bostonglobe.com [20]

Figure 2: **Examples of page snapshots loaded from IA.**

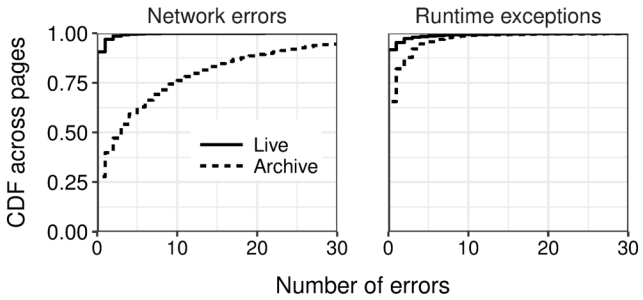


Figure 3: **Comparison of errors thrown during page loads from the web and from IA.**

2.1 Poor fidelity due to JS non-determinism

When a user loads a web page, scripts on the page often dynamically construct the URLs for many of the resources on the page. In doing so, JS execution can leverage various sources of non-determinism: client-side state (e.g., cookies, local-storage), client-characteristics (e.g., user-agent), random number generators, etc. When a user loads an archived page, these sources of non-determinism can potentially lead to a different set of resource URLs being requested compared to what was crawled by the archive. This, in turn, leads to two significant problems.

Failed fetches. First, IA returns a resource not found error (HTTP status code 404) for all resource URLs not stored at the archive, resulting in many archived pages rendering incorrectly. Figure 2 shows two examples of screenshots of page snapshots loaded from IA. In both cases, JS code on the page dynamically constructs the URLs of images to fetch by taking into account the screen size of the client loading the page. Since our client appears to differ from IA's crawler,[1] these pages end up being rendered incorrectly.

Runtime errors. Second, the execution of many scripts halts prematurely with runtime errors, which in turn leads to more resources going unfetched. We inspect the runtime logs generated by Chrome when loading the pages in $Corpus_{3K}$; specifically, the JavaScript console log and the network log. Figure 3 compares the number of errors seen in these logs during page loads from the web and when loading snapshots of these pages archived by the IA on the same day. Loading pages from IA results in more errors of both types. The total number of bytes that went unfetched because of these failed network requests cumulated to 5% and 45% of bytes on the median and 95th percentile page respectively.

2.2 High storage overhead

The more obvious downside of more JavaScript on web pages is that it increases a web archive's storage needs. To quantify this impact, we compute the total amount of storage required to store all the pages in $Corpus_{1M}$. Across all pages, we account for storing a single copy for every unique (resource URL, SHA-256 content hash) combination; IA applies similar deduplication to reduce storage overheads [23]. Despite the fact that scripts are often shared across pages (e.g., JavaScript libraries like jQuery are used by many sites and pages on the same site include a common set of scripts), JS accounts for 49% of all the bytes stored; resources of all other types (HTML, CSS, images, etc.) account for the remaining 51%.

Note that these numbers account for the size of textual resources such as HTML, CSS, and JS after compression. Also note that our corpus of a million pages appears to be large enough to approximate the utility of deduplication at scale because the fraction of total bytes accounted for by JavaScript plateaus after 750K pages.

Overall, the fact that scripts roughly double the amount of storage that a web archive needs to deploy is concerning because web archives are largely reliant on donations to cover their operational expenses [7]. For example, IA spends around $18 million dollars each year in operational expenses and attributes over 60% of its earnings to donations [14]. Needing to store more bytes per page means that an archive can store fewer pages for the same cost.

2.3 Downsides of alternate archival formats

To sidestep the shortcomings of IA that we have discussed thus far, a web archive could instead store and serve the end result of any page load, thereby preempting the need for clients to execute JavaScript.

Preserving post-load interactions. One such alternate archival format is to store a screenshot of the rendered page in the PNG or PDF format, as employed by private archiving institutions like Stillio [19] and PageVault [16]. However, many pages today enable users to interact with the content on the page, and storing screenshots of pages fails to preserve these post-load interactions [56]. Web developers enable such interactions by registering event handlers while a page is being loaded; these event handlers are triggered and executed when the user later interacts with the page. For example, modern pages often include carousels or sliders to display images and tabs to group information in separate categories; see, for example, the infographics on https://www.nytimes.com/interactive/2021/world/

[1] We look at the HTTP response headers of the archived resources to gather information about the client used by IA.

india-covid-cases.html. Event handlers are also used to enable users to navigate to other pages on the same site, e.g., the menu under the "Explore" button on https://www.coursera.org. Prior studies have shown that it is important to preserve such informational and navigational interactions even on archived pages [40].

We analyze the pages in *Corpus$_{3K}$* to determine how many contain interactions that should work on archived copies. Specifically, we load every page after instrumenting all scripts so that we can track all event handler registrations. We identify all event handlers which are associated with page elements whose attributes contain keywords such as menu, navbar, slider, carousel, dropdown, etc.; we consider 13 such keywords commonly associated with informational and navigational interactions. We find that 91% of the pages contained at least one such event handler.

Overhead of capturing JavaScript heap. Alternatively, client-local interactions enabled by event handlers could be preserved by storing a) every page's final rendered HTML, b) all resources referenced from this HTML (such as CSS and images), and c) the JavaScript heap, which stores custom, page-defined JavaScript state as well as native JavaScript objects [55]. However, modern browsers do not expose the entire JavaScript heap [43]; only the global scope of the heap is accessible using the "window" object. The closure scope, which is a non-global scope that is defined by any function and is accessible only by the nested functions that execute in that function's enclosed scope [51], is not accessible. This is a key roadblock because event handlers often access closure state; 47% of the pages in *Corpus$_{3K}$* contain at least one such handler (we describe how we perform the state tracking necessary to obtain this result in §4).

To access closure state, a web archive's crawler could statically analyze and rewrite the scripts on every page prior to executing them. However, we find that the combined overhead of performing the static analysis necessary to identify different scopes and running instrumented scripts inflates the time to crawl the median page in *Corpus$_{3K}$* by 2x; this overhead increases to 6x at the 99th percentile. Such computational overhead will significantly increase costs for a web archive crawling thousands of pages every second [64].

3 Overview

To overcome the adverse impacts of JavaScript on web archival, our high-level insights stem from two key differences between the loads of live and archived pages. In this section, we describe these differences and outline the challenges entailed in leveraging these differences.

3.1 Distinguishing properties of archived pages

No back-end origin server. Modern web pages include a range of functionalities which require communication with the page's origin servers, e.g., enabling users to post comments and having servers push updates to users while they are on a page. However, when a user loads an archived page snapshot, only that functionality on the page will work which can be served using the resources crawled when this snapshot was captured.

Limited sources of non-determinism. To deliver a dynamic user experience, many pages on the web adapt how they are rendered based on ① server-side state, ② client-side state (e.g., cookies, local storage), ③ client characteristics (e.g., user-agent, screen dimensions), and ④ "Date", "Random", and "Performance" APIs (we refer to these as *DRP* APIs for the sake of brevity). For example, after a script on a page fetches a JSON from the origin server, its subsequent control flow might depend on the contents of that JSON, which itself might be influenced by the contents of a client-side cookie. In loads of archived pages, the first two sources of non-determinism are absent: in response to the request for a particular resource in a specific page snapshot, a web archive will always serve the copy it fetched when crawling that snapshot; whereas, client browsers do not maintain any state across loads of archived pages.

3.2 Challenges

In order to leverage the above-mentioned differences to both improve page fidelity and reduce storage overhead in web archives, we need to answer several questions.

What are the causes of poor page fidelity? While some sources of non-determinism are absent in the loads of archived pages, the remaining sources – client characteristics, *DRP* APIs, and asynchronous execution of timer handlers and script fetches – still result in non-deterministic JS execution. Determining which of these factors is responsible for clients requesting different resource URLs than those crawled is key to eliminating failed resource fetches and the resultant runtime errors.

How to efficiently prune non-functional and unreachable code? In any page that it crawls, a web archive need not save any JS code that either relies on interactions with the page's origin servers or would never be executed in any load of the page (due to the absence of certain sources of non-determinism). One could potentially use methods like symbolic or concolic execution to perform reachability analysis and identify both unreachable code and non-functional code; the latter comprises code that is reachable from RPCs to origin servers. However, as reported in prior work [42, 49, 48], these methods for analyzing JS code are computationally expensive, requiring tens of minutes per page. Increasing the compute overheads of crawling to such a large extent would nullify any storage savings.

How to ensure code pruning does not hamper fidelity? While eliminating non-functional code reduces storage cost, doing so comes at the risk of inadvertently hurting fidelity. In particular, the code that is retained must function as it would if no code were discarded. Checking that any method identified for code elimination does preserve this property is

Goal	Observations	Section
Improve fidelity	APIs for client characteristics are the key cause for failed resource fetches	§4.1
	Differences in URLs due to *DRP* APIs can be resolved using server-side URL matching algorithms	
Prune non-functional code	Most of JS code which will not function on archived pages is in third-party source files, which can be identified based on their URLs	§4.2
	First-party scripts typically use third-party code cautiously, so that reliability of former is not dependent on availability of latter	
Prune unreachable code	*DRP* APIs typically have no impact on control flow	§4.3
	For event handlers associated with post-load interactions which work on archived pages, page state accessed is disjoint across handlers and user input does not influence control flow	

Table 1: **Overview of the main insights that influence our design of Jawa.**

non-trivial because browsers do not offer any APIs to extract runtime information that can be used to identify state dependencies between different scripts on any page.

3.3 Requirements

Based on all the considerations discussed thus far, we focus on three objectives.

- **High fidelity.** First, we seek to ensure that any archived page faithfully mimics the original page in two respects: 1) how the page is rendered, and 2) all functionality on the page which does not require communicating with the page's back-end servers works.
- **Low cost.** Second, we aim to enable a web archive to improve its coverage by reducing the amount of storage needed for any collection of page snapshots. In doing so, we seek computationally lightweight methods so as to minimize the cost overheads associated with maintaining the same rate of crawling pages as today.
- **Simplicity.** Lastly, our solutions must be simple to implement. In our discussions with the Internet Archive, they have emphasized that simplicity is key for any proposed changes to be viable in practice.

4 DESIGN

We describe our design of Jawa in three parts. We begin by describing how Jawa improves page fidelity by eliminating the sources of non-determinism which result in failed resource fetches while loading archived pages. Thereafter, we present the methods used by Jawa to identify what subset of crawled JS files need not be saved: first to eliminate non-functional code, and second to prune unreachable code while preserving post-load interactions. To enable Jawa's use at scale, the overriding principle that guides all aspects of our design is to minimize computational overheads by leveraging properties of JS typically found on the web; Table 1 provides an overview of our observations. Later (§7), we describe how a web archive which uses Jawa could potentially handle pages which do not satisfy these properties.

Analysis framework. Throughout this section, we use our custom JavaScript analysis framework (4.5K LOC) to study the properties of JavaScript found on pages in $Corpus_{3K}$. As in prior program analysis tools for JavaScript [49, 55, 38], our analysis framework first performs offline, static analysis of the JS in a page, converting each JS file into an abstract syntax tree (AST) representation. It then parses this AST to identify the different JS scope levels – local, block, closure, and global – and leverages this information to associate each JS variable to its corresponding scope. The framework also uses the AST to detect JS function invocations.

Building on these insights, our framework instruments pages with code that is triggered in each function invocation, and records the arguments to the function, all the closure and global scope variables read and written inside the function body, and the return value. Special care is taken to (1) record all accesses to the DOM, (2) track accesses of any global variable's properties via an alias, e.g., *"var a = window"* followed by a read of *"a.innerHeight"*, (3) identify DOM elements with registered event handlers and the corresponding handler functions, and (4) monitor and control the return values of browser APIs such as *"navigator.userAgent"*.

4.1 Improve fidelity by eliminating failed fetches

To ensure that users do not encounter failed resource fetches when they load archived pages, a web archive could rewrite every stored page to ensure that, when the page is loaded, the flow of execution and the return values of all browser APIs match those seen when the page was crawled.[2] If a web archive were to eliminate sources of non-determinism in this manner, we observe that fixing the schedule of execution cannot result in any loss of functionality; after all, developers of pages have no control over the client-side schedule of execution of asynchronous scripts. However, a page's developer can indeed ensure that code on the page behaves differently based on the results from browser APIs. Therefore, we seek to understand the impact of these APIs on resource URLs and eliminate only those sources of non-determinism which result in failed fetches during loads of archived pages.

Impact of different sources of non-determinism. We measure the impact of each source of non-determinism as follows. We first load our locally stored copies of all pages in $Corpus_{3K}$ with a desktop client. We then reload these pages mimicking a different client ("iPhone 6"). Mimicking a different client allows us to exercise different values of most

[2] Alternatively, a web archive could crawl every page under all possible combinations of non-determinism. Doing so is not only impractical, but would dramatically inflate compute and storage overheads.

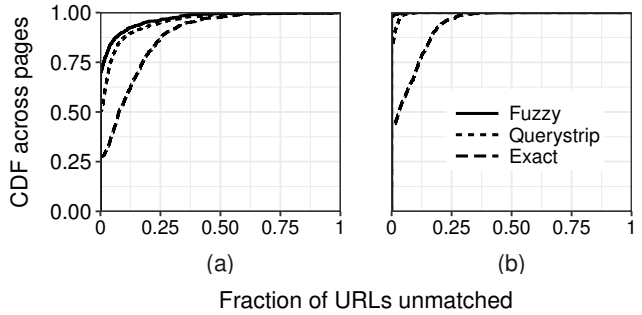


Figure 4: **For every page in *Corpus$_{3K}$*, fraction of resource requests which cannot be matched with any crawled resource. The impact of different URL matching algorithms is shown when the sources of non-determinism are (a) APIs for client characteristics as well as *DRP* APIs, and (b) only *DRP* APIs.**

client characteristics, such as user-agent, screen dimensions, and OS. We reload all pages once more, this time matching the client characteristics used in the original load.

On 72% of pages, at least one different resource URL was requested in the second load compared to the first load; these two loads differ in the values for both APIs for client characteristics and *DRP* APIs. Whereas, when comparing the third load to the first, which differ only with respect to *DRP* API values, the corresponding fraction was 52%. Note that, in both cases, even one failed resource fetch can have a cascading effect, resulting in many other resources going unfetched.

Variance in resource URLs due to non-determinism results in failed network fetches only if a web archive (like IA) expects requests from clients to specify URLs which are identical to the ones crawled. However, across loads of a page, if the same resources are being requested using different URLs, it might suffice for the web archive to employ a better algorithm to match URLs requested to those crawled.

To check if this is the case, we consider two URL matching algorithms used in prior work: ① *querystrip*, where the query string in any URL (i.e., the portion of the URL beyond the delimiter '?') is stripped before initiating a match [57], and ② *fuzzy matching*, which leverages Levenshtein distance [45] to find the best match for any given URL [26]. Querystrip relies on the fact that query strings are typically used for updating server-side state, and they do not influence the content of the response. Fuzzy matching accounts for cases where non-determinism across loads results in simple string transformations of the URLs for the same resources. In any page load, we match URLs in the order they are requested, and we match any requested URL against those crawled URLs that have not already been matched.

Figure 4(a) shows that, on many pages, a significant fraction of URLs were unmatched with both algorithms, when APIs for client characteristics were a source for diverging URLs. This is because, when client characteristics differ, often the *number* of resources fetched on the same page changes. For example, www.nytimes.com fetches the JavaScript file *player-embedded.js* on mobile clients to enable video players, whereas it fetches no such scripts on desktop clients.

Digging deeper into *DRP* APIs. In contrast, when *DRP* APIs are the only source of non-determinism, Figure 4(b) shows that either URL matching algorithm suffices to eliminate almost all failed resource fetches. However, this might be the case only because we compare two loads of every page, and the return values of *DRP* API invocations did not sufficiently differ to have an impact.

To capture the effects of all possible return values of *DRP* APIs, we turn to concolic execution [37, 61, 42], a variant of symbolic execution which executes programs concretely (rather than symbolically) while ensuring complete coverage of all control flows. We modify a prior concolic execution tool [42] to only track control flows influenced by *DRP* APIs. We then randomly sample 300 pages from *Corpus$_{3K}$* because it takes around 20 minutes per page with this tool. On all pages, *DRP* APIs had no impact on control flow. Thus, comparing any two loads of a page suffices to examine the divergence in URLs across loads due to these APIs.

Takeaways. These results influence our design of Jawa in two ways. First, we instrument all scripts on any page so that, when clients execute these scripts, all APIs for client characteristics return the same values as when the page was crawled. Compared to a thin-client model where a web archive serves requests for pages by executing page loads on behalf of users [26], our approach of letting users execute page loads on their devices reduces server-side overheads. Second, we do not need to account for any differences across loads in *DRP* APIs because the impact of these differences can be accounted for with server-side matching of requested URLs to crawled URLs.

Note that we choose to patch all invocations of client characteristic APIs, and not just the ones which influence the URLs fetched. This is because, even if a particular invocation of an API does not impact which URLs are fetched, it can impact the reachability of code which assumes that state dependent on the client's type has been setup earlier in the page load. Hence, if different API invocations return inconsistent values, this could exercise code which accesses uninitialized state, resulting in runtime errors.

4.2 Pruning non-functional code

We now turn our attention to reducing the storage overhead of JavaScript on web archives. Jawa's crawler uses two complementary approaches to take advantage of the two previously mentioned properties which distinguish archived page snapshots from pages on the web. The key consideration in both cases is to ensure that pruning any JavaScript code does not affect the execution of the remaining code.

Characteristics of non-functional code. Our first approach for pruning JavaScript code is based on two observations about the code which will not work on archived copies of pages, i.e., code which relies on clients interacting with

origin servers. First, on a typical page, we find that most of such code is compartmentalized into a few files, rather than being evenly spread across all JavaScript source files on the page. As we will show later, these files do not contain any code that is worth preserving. Second, functionality which will not work on archived pages is largely implemented by third-party scripts. Even though some of the functionality which relies on communication with origin servers (e.g., intra-site search, login) is implemented by the first-party origin, we only focus on discarding third-party files, for reasons discussed shortly.

The implication of these observations is that, to identify most of the non-functional JavaScript code in archived pages, it is unnecessary to perform any complex code analysis. Instead, it suffices to assemble and use a "filter list" which captures the features distinctive to the URLs of scripts containing non-functional code; when crawling pages, a web archive would simply have to discard (and not even fetch) any script whose URL matches the filter list.

For example, via manual analysis of the URLs of all scripts seen in $Corpus_{1M}$, we assemble a filter list comprising 45 rules. We consider those script URLs which are included on many pages. For each such popular script, we first visit the domain on which the script is hosted to understand the services offered by that domain. In cases where a domain hosts scripts of many kinds, some of which are important to retain even on archived pages, we examine the script's content to determine its utility.

Every rule in our list matches URLs at one of three granularities: 1) domain, i.e., filter any file hosted on that domain (e.g., *"zephr.com"* enables support for user subscriptions), 2) file name, i.e., filter scripts if the file name matches, regardless of the domain hosting the script (e.g., *"jquery.cookie.js"* is used for cookie management), and 3) URL token, i.e., filter scripts if a specific keyword appears anywhere in their URL (e.g., *"pagesocial-sdk"* and *"recaptcha"*).

Recall that $Corpus_{1M}$ comprises page snapshots crawled from the Internet Archive, which already discards resources that users often block on the live web, e.g., ads. In contrast, our filter list aims to prune scripts which implement functionality that is important to preserve on the live web, but will not work on archived copies. Moreover, since a few popular third-party service providers are used by the vast majority of websites [46], we find that we only need to add 6 rules to our filter list to account for pages on 300 additional sites beyond the 300 sites included in $Corpus_{1M}$.

Filtering has no impact on fidelity. Discarding a subset of the JS files on a page might, however, break the execution of code in files that are retained. Therefore, we study the impact of filtering along two dimensions: 1) visual (i.e, does the page look the same?), and 2) functional (i.e, are post-load interactions that will work on archived pages unaffected?)

We load every page in $Corpus_{3K}$ with and without filtering enabled. We take a screenshot after every page load.

```
<script src="https://js.sentry-cdn.com/7bc8b.min.js" </script>
<script>
        if (window.Sentry) {
            window.Sentry.onLoad(function() {
                window.Sentry.init({
                    maxBreadcrumbs: 30,
                environment: 'prd', });
            });
        }
</script>
```

Figure 5: **Code snippet from www.nytimes.com where the main frame first fetches a third-party JavaScript file hosted on www.js.sentry-cdn.com and then cautiously invokes a function from it inside an if condition.**

Leveraging our JavaScript instrumentation described earlier, we also 1) identify all event handlers registered during each page load, 2) trigger all event handlers after the page load completes, and 3) track all values read or written from the JavaScript heap and DOM by these handlers.

First, when we compare the screenshots for every page with and without filtering, we observe that these screenshots differ in the value of at least one pixel for 109 of the 3000 pages in $Corpus_{3K}$. Upon manual examination of these 109 pages, we find that all differences are either due to animations or because *DRP* APIs result in a different timestamp on the page. Second, for all event handlers registered by the unfiltered files, we find 35 pages on which at least one value accessed by at least one of these event handlers differed across loads with and without filtering. Again, these differences were not consequential: they were due to differences in timing information, e.g., some event handlers log the times at which their execution starts and ends.

A key reason for these positive results, which show that Jawa's filtering has no impact on the fidelity of the code retained, is our explicit choice to only consider third-party source files for filtering. On the one hand, most third party scripts are self-encapsulated, i.e., the code in these files only interacts with itself or the files it subsequently fetches. On the other hand, as shown in Figure 5, first-party scripts typically invoke third-party code cautiously, so that the former is unaffected in the off chance that the latter fails to be fetched.

Note that one cannot simply eliminate *all* third-party scripts; that would render dysfunctional many post-load interactions which do work, and are important to preserve, on archived pages. As we show later in our evaluation (§6), while discarding files which match our carefully curated filter list enables significant storage savings, doing so preserves all navigational and informational interactions.

4.3 Prune unreachable code

In the Javascript files which do not match Jawa's filter list, many lines of code will never be executed in *any* page load. This is because 1) some sources of non-determinism are absent in loads of archived pages (§3.1), and 2) Jawa elim-

inates non-determinism caused by asynchronous execution and APIs for client characteristics (§4.1). Furthermore, we found that *DRP* APIs have no impact on control flow. Yet, identifying all reachable code remains challenging: beyond the code executed while crawling the page, we also need to retain the code for users' post-load interactions.

Challenges in preserving interactions. Post-load interactions are enabled via event handlers which are registered while a page is being loaded. Every event handler is associated with a specific DOM node on the page, and is bound to a specific action that would trigger the handler, such as a click, scroll, mouse hover, etc.

The code that is exercised when an event handler on a page is invoked can vary as a function of a) the order in which the user interacts with different elements on the page, b) the inputs that the user provides to these events [30], and c) the return values of browser APIs. It is easy to see how the latter two can impact code reachability, e.g., in response to a search query, the number of search results can influence certain client type-specific UI features, such as the option of splitting the results across multiple pages. The order in which events are triggered can impact the execution of some handlers if the state read (from the DOM or JavaScript heap) by one handler could have been written to in a prior invocation of this or another handler. In particular, since we only care about identifying reachable code, only read-write dependencies which impact branch conditions are of interest.

We analyze the impact of these sources of non-determinism on the event handlers found on pages in $Corpus_{3K}$. We capture the state accessed by event handlers as described earlier in §4.2. For each event handler on a page, we check whether there is a read-write state overlap with itself or with any other event handler on the page, and if there was an overlap, whether this state is used in a branch condition. We also identify all handlers which accept user inputs; these include mouse events (e.g., click, mouseover, mouseon), keyboard inputs (e.g., keyup, keydown), and text inputs (e.g., "INPUT" or "FORM" DOM nodes). When we invoke each such handler, if a branch statement is executed, we conservatively conclude that the handler's inputs could impact the control flow of the handler.

On each page, we compare two sets of event handlers: those which work on the live version of these pages, and the subset which will work on archived copies. The former set comprises all handlers registered when we load the page without filtering. We identify the latter set of handlers by loading every page with Jawa's filtering enabled, and ignoring handlers which interact with origin servers (i.e., they are registered on either "INPUT" or "FORM" DOM nodes with a corresponding "action" attribute). On the median page, 14 event handlers work on the live page and 7 on the archived copy. At the 90th percentile, the corresponding numbers are 170 and 44.

Impact of order. On 40 of the 3000 pages in $Corpus_{3K}$, at least one pair of handlers that work on the live page had a read-write dependency which could affect the control flow of one of these handlers. In contrast, we found no such case when focusing on the handlers which work on archived pages. This stark difference is because dependencies between handlers arise on live pages predominantly due to analytics, e.g., handlers registered with certain DOM nodes update locally maintained state to track if the user interacted with those nodes; when the user navigates away from the page, another handler reads this state and sends this information to back-end servers if the user did interact with those DOM nodes.

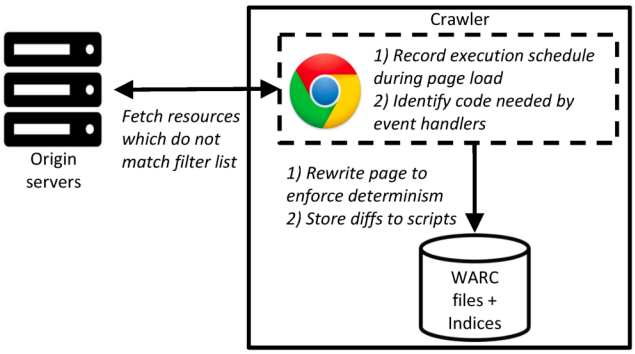


Figure 6: **High-level overview of Jawa.**

Impact of user input. Across all pages, none of the event handlers which work on archived copies read inputs which influenced branch predicates. Whereas, when we loaded all pages without filtering, 1134 of the 3000 pages had at least one handler which interacted with back-end servers. In such cases, the responses from servers could potentially impact what code gets executed on the client.

Impact of browser APIs. As discussed earlier (§4.1), Jawa eliminates non-determinism caused by APIs for client characteristics. That leaves *DRP* APIs. Among the handlers that work in an archived context, 449 of the 3000 pages had at least one handler which invoked an API from either "Date" or "Math.random". All the invocations of "Math.random" APIs were due to the jQuery library assigning unique identifiers to elements inside its `ElementSelector` function [15]. Whereas, the "Date" API was used only for logging the start and end time of handler executions. Thus, in all of these cases, *DRP* APIs did not impact the reachable code for any event handler.

Takeaways. These results demonstrate why prior work which aims to identify code reachable by event handlers performs complex JavaScript program analysis [30, 59]. In contrast, we find that no source of non-determinism impacts the code executed by handlers which work on archived pages. Therefore, on any page, to identify the code necessary to retain for post-load interactions to work, it suffices for Jawa to invoke every handler once and save the code that is executed.

4.4 Summary

Put together, our observations on the differences between loads of archived and live pages enable Jawa to use a fairly simple methodology to crawl and save pages, as shown in Figure 6. For every page that it crawls, Jawa fetches all those resources which do not match its filter list. For the remaining files, it ① injects code to identify what code was executed during the page load and in what order, and ② triggers every registered event handler using default input values (e.g., the default x and y coordinates for a mouse click event is 0,0) and identifies the code executed. Finally, it stores those portions of the page that are exercised in either step above. It instruments the retained code so that, when users load the page, their browser follows the same execution schedule and uses the same client characteristics.

5 IMPLEMENTATION

Implementing a web archive involves several considerations which are outside the scope of this paper, e.g., distributing data across servers, detecting and coping with hardware failures, etc. Our implementation focuses on the aspects of a web archive addressed by Jawa (Figure 6), namely crawling and storing page snapshots. We also describe the impact of Jawa's design on serving page snapshots to users.

5.1 Crawling pages

When crawling a page, Jawa's crawler (1.2K LOC) uses a Node.js based man-in-the-middle proxy to interpose on all requests/responses. The proxy uses the Esprima [9] and BeautifulSoup [4] libraries to instrument JavaScript and HTML files as they are fetched. Jawa references the filter list for every outgoing request and, using regular expression matching, blocks the request for any resource whose URL matches any of the rules in the filter list. For all the remaining resources fetched, Jawa selectively instruments JS files prior to their execution. This instrumented code, upon execution, enables Jawa to 1) interpose on all browser APIs, 2) track the subset of JS code executed (in terms of JS functions), and 3) helps enumerate all event handlers registered on the page. The instrumentation overhead incurred by the crawler is significantly lower compared to when tracking all state accesses (§4).

5.2 Storing page snapshots

For every page that it crawls, Jawa saves only a subset of the JavaScript code on that page. Consequently, when the same JavaScript file (e.g., a library) is included on many pages, it is often the case that different subsets of this file need to be stored as part of different page snapshots, thereby preempting simple file-level deduplication, as used by the Internet Archive today [23].

Our solution is to store every unique file as a set of partitions; each partition represents a different disjoint subset of the file: from a specific start byte offset to an end byte offset. When Jawa crawls a new page snapshot, for every JavaScript file crawled that is not filtered, it identifies the subset of code in this file relevant for this snapshot. It then looks up the crawl index (Table 2) to determine if this subset is already covered by the byte ranges in this file that have previously been stored. The crawler creates new WARC records for portions of the file that have not been previously stored and appends new entries to the crawl index. The crawl index is processed asynchronously to produce the serving index (like is the case today with Internet Archive).

Crawl index		
	Key	*Value*
IA	URL	List of (content hash, WARC file ID) tuples
Jawa	(URL, content hash)	List of (start byte offset, end byte offset, WARC file ID) tuples

Serving index		
	Key	*Value*
IA	(URL, timestamp)	(WARC file ID, byte offset)
Jawa	(URL, timestamp)	List of (WARC file ID, byte offset) tuples

Table 2: **Comparison of indices maintained by IA and Jawa.**

5.3 Serving page snapshots

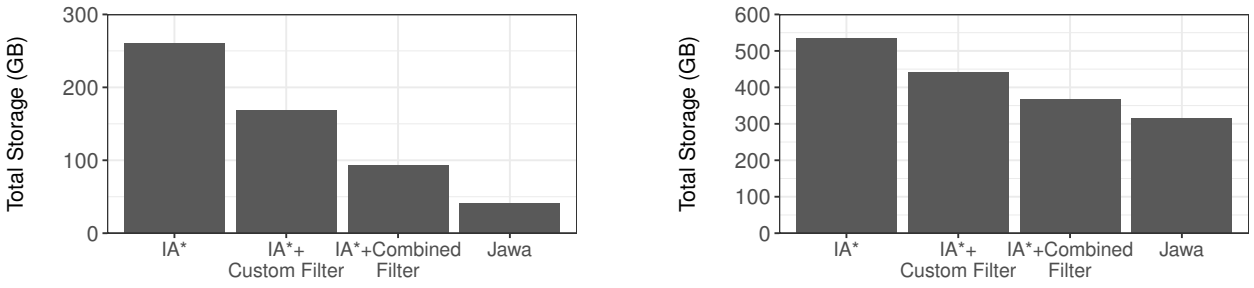
The implication of storing any JavaScript file's contents as above is that, when a client requests for a file while loading a page snapshot, one does not know which of the partitions stored for this file are relevant for this particular snapshot. Instead, a web archive which uses Jawa can return the union of all stored partitions for the requested JavaScript file; after all, the portion of the file needed for any snapshot is a subset of the stored partitions. Since the size of this union is at most equal to the size of the original file, clients will have to fetch no more bytes than they do today.

6 EVALUATION

We evaluate Jawa with three metrics: storage (to store crawled resources and to store indices), fidelity (similarity of archived page snapshots to the corresponding original pages) and performance (both for crawling and serving). In all cases, we compare against the corresponding techniques currently in use by the Internet Archive (§2), which we refer to as IA*.[3] In some cases, we also break down the utility/overhead of each of Jawa's components. The key findings from our evaluation are as follows:

- Jawa reduces the storage needed for our corpus of 1 million page snapshots by 41%. This reduction stems from Jawa discarding 84% of JavaScript bytes.
- Despite this significant reduction in storage, on a random sample of pages, all event handlers that one would expect to function on archived pages continue to work.
- When we mimic loads of archived pages from IA, at least a quarter of resource fetches fail on more than 10% of pages.

[3] IA* refers to us mimicking the techniques used by IA.



(a) JavaScript resources (b) All resources

Figure 7: **Total storage necessary to store corpus of 1 million page snapshots.**

Whereas, on over 99% of pages, Jawa eliminates all failed network fetches and ensures that the set of resources requested from the archive match those crawled.

- Crawling throughput with Jawa improves by 39%, thanks to our use of lightweight techniques for code analysis and filtering of JavaScript files.

6.1 Storage

6.1.1 Storage for resources.

To begin, we consider the total amount of storage needed to store the resources in our $Corpus_{1m}$ corpus. We crawl all of these page snapshots from IA using our crawler (§5). On each page, Jawa's crawler only fetches third-party JavaScripts which do not match its filter list. Apart from our manually curated filter list for pruning code which will not function on archived pages, we also leverage the open-source filter list from EasyList [8], which is widely used by many browser extensions to identify ads and analytics. In every script that it does fetch when crawling a page snapshot, Jawa's crawler identifies the subset of code necessary for this snapshot and stores the portion of this subset that is not covered by the subsets of this file previously stored.

Figure 7(a) shows that Jawa stores 40 GB of JavaScript across the 1 million pages, a reduction of 84% compared to IA*. Of course, to store the entire corpus, all resources on every page snapshot need to be saved, not only JavaScripts. For resources other than scripts (images, CSS, HTML, fonts), Jawa offers no storage benefits; it stores them exactly as IA*. Yet, we see a 41% reduction in total storage: 535GB with IA* to 314GB with Jawa (Figure 7(b)). This is because, as seen earlier in §2.2, JavaScript files account for 49% of all the bytes across all pages, even after file-level deduplication. Since 63% of the more than 140 PB of data stored by IA is devoted to web page snapshots [12, 13], we estimate that Jawa can reduce IA's storage needs by 35 PB.

Sources of storage benefits. Storage savings enabled by Jawa stem from a combination of not storing filtered files and pruning unreachable code. When we break down the impact of the filter lists we use, Figure 7(a) shows that our custom filter list alone reduces the total amount of JavaScript saved by 36%, and EasyList's rules result in a further reduction of 28%. Jawa also significantly reduces storage needs by eliminating unused code: the difference between the two right most bars in Figure 7.

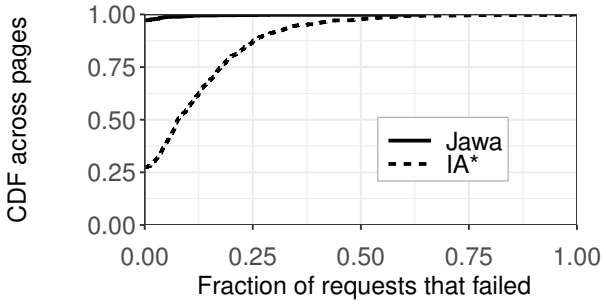
6.1.2 Storage for indices

In addition to storing crawled resources, both IA* and Jawa also need to store the crawling and serving indices (Table 2). The former enables the crawler to not store duplicate content, whereas the latter enables lookups of requested resources when serving page snapshots. For our corpus of 1 million page snapshots, we find that size of both indices is marginally smaller (15%) with Jawa than with IA*. First, for most script files, Jawa ends up having to store a single WARC record; for such files, after the first time a subset of the file's code is stored, all subsequent page snapshots which include the same file end up needing the same subset. Second, the increase in index entries for other files (for which multiple subsets end up being stored) is offset by the elimination from the index of filtered files.

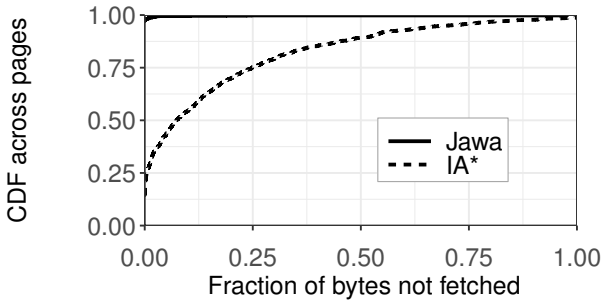
6.2 Fidelity

To evaluate Jawa's preservation of page fidelity, we crawl all 3000 pages in $Corpus_{3K}$ from the live web. We perform these crawls on a desktop, once with Jawa's crawler, and once without using any of its methods. We then load these pages from the two local copies, mimicking a different client ("iPhone 6"). When using page snapshots saved by Jawa, we match requested URLs to crawled URLs after stripping query strings.

Resource fetches. We first evaluate Jawa's impact on fidelity by examining the discrepancy between the set of resources stored for any snapshot and the set of resources fetched by a client when it loads that snapshot. Figure 8(a) shows that, while 7% of network requests return a 404 on the median page in loads of IA*, this fraction drops to 0% with Jawa. On the 95th percentile page, the corresponding fractions are 36% with IA* and 0% with Jawa. Consequently, Figure 8(b) shows that, while 10% of stored resources are not fetched on the median page when mimicking loads from IA, this fraction drops to 0% with Jawa. On the 95th percentile page, the corresponding fractions are 75% with IA* and 0% with Jawa.



(a)



(b)

Figure 8: **When snapshots of 3K pages are served, (a) number of resources requested by client which are not stored, and (b) fraction of resources stored for a snapshot which are not fetched by the client.**

Visual analysis. To check if the pages served by Jawa are identical to the ones it crawled, we take a screenshot of every page both when crawling it and when we reload it from our local copy. We then compare every pair of screenshots to check if the value of every pixel matches. Apart from the visual differences accounted for by animations and non-determinism in 54 pages, both screenshots matched exactly for every other page when using Jawa. Since loads of IA* do not patch APIs for client characteristics, differences in screen dimensions between clients make it moot to compare screenshots.

Interactions. Finally, to evaluate Jawa's impact on post-load interactions, we randomly sample 150 pages. For each page, we load the versions that would be served by IA* and by Jawa. To isolate the impact of Jawa's techniques, we also consider an intermediate design point (Only filter) where we only use Jawa's filtering but do not prune unreachable code.

We categorize all event handlers on every page into three types: 1) navigational, i.e., they help in navigating either to a different page (e.g., a navigational bar) or within the page (e.g., a scroll-to-bottom button), 2) informational, i.e., they help make more information available (e.g., carousels or tabs), and 3) transactional (e.g., login or post buttons). On archived pages, transactional event handlers will not function. So, on each of the 150 sampled pages, we manually trigger all event handlers that belong to the first two categories. All 124 navigational interactions and 100 informational interactions worked as expected in all three loads: IA*, Only filter, and Jawa. Key to preserving these post-load interactions are Jawa's carefully curated filter list for discarding non-functional code, and its methods for identifying and retaining all reachable code. In contrast, if we discard all third-party files or if we use Jawa's filter list but save only the functions registered as handlers, then only 42% of these interactions work in the former case and 10% in the latter.

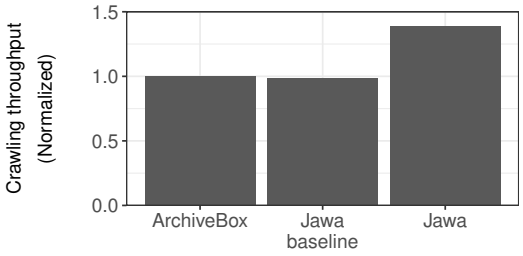


Figure 9: **Comparison of crawling throughput, normalized to that offered by ArchiveBox.**

6.3 Performance

Crawling throughput. IA's production crawler is not public to the best of our knowledge. Therefore, we turn to two open-source crawlers: Brozzler [5] and ArchiveBox [3]. Brozzler is operated by IA, and used alongside their production crawler. Whereas, ArchiveBox is a very active and commonly used crawler by individual archivists (over 12K stars on GitHub). We find that Brozzler is 20% slower than ArchiveBox because of the latter's more efficient implementation of their headless Chrome interface. We also note, that on a server with 32 cores and 128 GB RAM, we were able to crawl 5000 URLs in 15 minutes with ArchiveBox. With this crawling throughput, IA would need to dedicate 900 such servers for crawling pages, which is comparable to the number of servers they currently claim to use [11]. Therefore, we evaluate Jawa against ArchiveBox.

Figure 9 shows that Jawa's crawler offers throughput comparable to Archivebox when all of Jawa's techniques are disabled (Jawa baseline). Enabling all the methods in Jawa's design increases our crawler's throughput by 39%.

To breakdown the overheads, we measure the latency of each of the techniques used by Jawa's crawler in isolation, namely 1) filter: filtering JavaScript files, 2) code injection (CI): instrumenting the code in fetched scripts, 3) dynamic tracking (DT): dynamically tracking code execution and event handler registration, and finally 4) event triggering (ET): invoking event handlers and capturing the code executed. Figure 10 shows that not having to fetch filtered scripts completely offsets the overheads of all other techniques. Not only does Jawa's crawler not fetch any scripts which match its filter list, but all the resources that would have been fetched by the filtered files also go unfetched; this latter set of files often do not match the filter list.

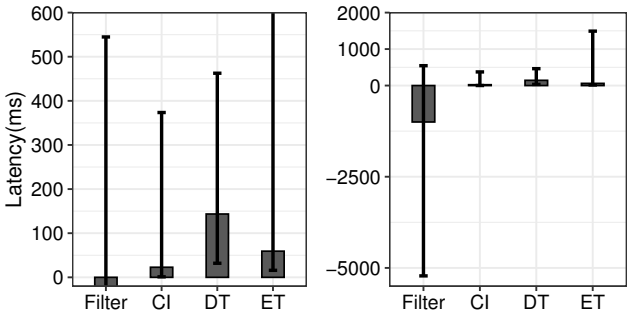


Figure 10: **Benchmarking the overhead of techniques used in Jawa's crawler. Bars plot median across pages, and whiskers plot 10^{th} and 90^{th} percentiles. Graph on the left zooms in on the yrange 0 to 500ms in the graph on the right.**

Index	I/Os per page with IA*		Reduction in I/Os per page with Jawa	
	50^{th} %ile	90^{th} %ile	50^{th} %ile	90^{th} %ile
Crawling	3	15	1	5
Serving	41	107	1	3

Table 3: **Writes on crawling index and reads on serving index; values shown for 50th and 90th percentile page on median site.**

Jawa also impacts crawling throughput by requiring more writes to the crawling index because, unlike IA*, it spreads the code in some script files across multiple WARC records. We cannot quantify the performance impact of doing so since our setup does not match a production archive like IA. However, we can quantify the number of additional writes that Jawa performs to the crawl index, compared to IA*. Table 3 shows that the number of writes to the crawl index *decrease* with Jawa; due to filtering, fewer files are crawled.

Serving performance. When serving page snapshots, Jawa's only overhead is in needing to potentially lookup multiple WARC records in order to respond to a request for a JavaScript file. We find that page load times on IA's Wayback Machine are proportional to the number of resources on the requested page snapshot, or equivalently, the number of WARC records that IA needs to lookup to serve the snapshot. Therefore, as a proxy for estimating Jawa's impact on user-perceived performance, we examine the increase due to Jawa in the number of WARC records read when serving page snapshots. Table 3 shows that the number of index lookups decrease with Jawa; again, thanks to filtering, a client has to fetch fewer files per snapshot.

7 Verifying Page Properties

Jawa's methods for pruning non-functional and unreachable code are based on three properties that we found to be true on archived web pages:

- *DRP* APIs have no impact on control flow
- Discarding third-party JavaScript files which match a manually curated filter list has no impact on fidelity
- For post-load interactions which work on archived pages, the event handlers which power them do not have read-write dependencies that influence branch conditions

All of these observations are rooted in our empirical analysis of a variety of web pages in *Corpus$_{3K}$*: 9 internal pages and 1 landing page in each of 300 sites, which span a wide range of rankings among Alexa's top million sites. However, we recognize that not all pages may abide by these properties. For example, consider a page which shows the time until a deadline and switches the font color when the time remaining is below a threshold; such a page would violate the first property listed above.

To handle such cases, we observe that web archives do not crawl every page just once; they repeatedly recrawl pages over time in order to capture changes to every page's content. For any given page, in some crawls of the page, a web archive can disable all of Jawa's methods and check if the properties expected to be true indeed hold on this page. For example, like the analysis we performed (§4), the web archive can instrument scripts to track state accesses, and then examine dependencies between event handlers and between files which do or do not match the filter list. It can also perform concolic execution to verify that *DRP* APIs have no impact on control flow.

The key to restricting the compute overheads of these heavyweight analyses is to run them on a sample of snapshots. To determine the sampling rate, a web archive can leverage properties that are stable across a page's snapshots. For example, upon analyzing all of IA's snapshots for 300 randomly chosen pages, we observe that the median page has the same number of runtime errors for an average of 53 snapshots. Therefore, once in every 53 crawls of any of these pages, a web archive can disable filtering and check if the number of runtime errors matches prior crawls where filtering had been used. If there is a mismatch, the web archive can disable the use of filtering for this page going forward. Since Jawa serves any JavaScript file to users as the union of all partitions of this file stored across crawls (§5), disabling filtering in one crawl of the page will also benefit all prior crawls of that page.

8 Discussion

How future proof is Jawa? In the immediate future, recent trends [18] indicate that the amount of JS on pages will continue to increase, making it important for web archives to adopt Jawa's techniques for pruning JS and for eliminating fidelity issues due to the non-determinism introduced by JS. In the long term, we expect that the principles that dictate Jawa's design will continue to hold: to serve pages with high fidelity, 1) archives must account for non-determinism, and 2) a large fraction of JS can be discarded with no risk.

Optimize already archived pages. Jawa's simple techniques make it highly amenable to be used with pages that have already been archived. First, a web archive can sig-

nificantly reduce its storage needs by discarding all JS files that match Jawa's filter list. Second, the web archive can rewrite the HTML of every archived page to include a custom script which will enforce the same client characteristics as the crawler when users load the page. The only aspect of Jawa that would be hard to use on already archived pages is the elimination of unreachable code, as that requires invoking all event handlers on every page.

9 Related work

Impact of JavaScript on web crawlers. Prior work has shown that it is important for web crawlers to execute JavaScript when crawling pages, both in the context of web archives [31, 32, 33] and web search engines [1], else many important resources on a page will often go uncrawled. Our work highlights that, due to the non-deterministic execution of JavaScripts, archived pages often have poor fidelity even when pages are crawled using a browser which executes all scripts on every page.

Beyond executing JavaScripts while crawling a page, systems like Conifer [6] also save all resources on the page that are fetched while the user is interacting with the page. However, such systems are designed for private web archival, i.e., a user saves a page and its constituent resources for the user's own personal use later. If users load a page archived by a different user using a different device/browser, they will face the same fidelity issues seen on the Internet Archive.

Coverage of web archives. Many measurement studies [27, 28] have demonstrated that web archives are far from comprehensive in archiving all pages on the web. Prior work [50, 41] has attempted to address the incompleteness caused due to large portions of the web not being openly available (e.g., behind paywalls) and requiring user logins (e.g., social media). In contrast, we seek to enable web archives to improve their coverage by reducing the costs associated with archiving any corpus of pages; thereby, for the same budget, a web archive can crawl and save more pages.

Supporting bulk processing of archives. Jawa focuses on enabling web archives to support the use case where users load individual page snapshots and interact with them. Alternatively, web archives are used by researchers to perform large scale analyses of historical information. Xinyue et al. [64] demonstrate the performance penalties of the WARC format for such batch processing workloads, and many systems [47, 2, 39] have been developed to enable programmatic analysis of large corpuses without needing to access each individual resource on every page.

JavaScript record and replay systems. A number of prior systems [29, 52, 60] enable users to record and replay JavaScript execution, both in the context of browsers [29] and independent JavaScript programs [60]. These record and replay tools are critical for debugging JavaScript based errors. Therefore, to ensure high fidelity replay, all of these systems identify and patch all sources of non-determinism to match the recorded version. In contrast, we analyze the individual impact of each source of non-determinism on the URLs fetched and patch them accordingly.

Code reachable through event handlers. JavaScript testing tools automate the process of testing by dynamically constructing test cases to achieve maximum code coverage. A key part of this process is identifying all code that can be potentially executed by event handlers. Doing so requires heavyweight symbolic execution analysis [42], or exhaustively going through all possible orders and inputs [30]. Jawa leverages the differences between archived and live web pages to simplify this analysis by only needing to use the trace from a single execution.

Program analysis on the web. JavaScript on the web has been notorious for various kinds of security, privacy and performance issues. A large body of prior work focuses on addressing such issues by relying on sophisticated program analysis techniques [63, 66]. Such techniques, however, incur a high computation cost. This is why, in solutions for optimizing web performance [42, 54, 49] which use computationally expensive JavaScript analysis techniques, web servers perform such analysis in the background to mitigate the impact of their overheads. For archival systems, even if crawled JavaScript resources are processed offline, the cost for computationally heavyweight processing is not sustainable. Hence, Jawa employs lightweight approaches, rooted in properties of JavaScript on the web.

Dead code elimination on the web. One way to optimize web performance is to eliminate dead code (i.e., code that is never reachable) from resources such as JavaScript and CSS. Tools [17, 25] which do so using static analysis are widely used. We observe that, in archived pages, a significantly greater fraction of code is potentially unreachable, since many sources of non-determinism (e.g., variation in client state and server responses) are absent. Jawa exploits this property to provide significant storage savings.

10 Conclusion

Since when the Internet Archive began operating in the late 1990s, a marked change on the web has been the increased use of JavaScript. In this paper, we shined light on two significant problems caused by this change: broken rendering of archived pages, and petabytes of storage wasted on JavaScript which will either be non-functional or never be used. Our design of Jawa addresses these problems while emphasizing low overhead on both crawling and serving pages. As a result of our work, web archives will be able to archive many more pages than they can today for the same cost and ensure that archived pages more closely approximate their original versions.

Acknowledgements: We thank the anonymous reviewers and our shepherd, Philip Levis, for their valuable feedback.

REFERENCES

[1] https://developers.google.com/search/docs/advanced/javascript/javascript-seo-basics.

[2] Archive unleashed. https://github.com/archivesunleashed/aut.

[3] Archivebox. https://github.com/ArchiveBox/ArchiveBox.

[4] BeautifulSoup. https://pypi.org/project/beautifulsoup4/.

[5] Brozzler. https://github.com/internetarchive/brozzler.

[6] Conifer. https://conifer.rhizome.org/.

[7] Donate to the Internet Archive! https://archive.org/donate/.

[8] EasyList. https://easylist.to/.

[9] Esprima. https://esprima.org/.

[10] HTTP Archive: State of the web. https://httparchive.org/reports/state-of-the-web#bytesTotal.

[11] IA infrastructure. https://archive.org/details/jonah-edwards-presentation.

[12] Inside wayback machine. https://thehustle.co/inside-wayback-machine-internet-archive/.

[13] Internet archive. https://www.archive.org/about/.

[14] Internet Archive tax return. https://projects.propublica.org/nonprofits/organizations/943242767.

[15] jQuery element selector. https://api.jquery.com/element-selector/.

[16] Page-vault. https://www.page-vault.com/solutions/.

[17] Prepack. https:/www.prepack.io.

[18] State of JavaScript. https://httparchive.org/reports/state-of-javascript.

[19] Stillio. https://www.stillio.com/.

[20] The Boston Globe: Internet archive's copy from August 2, 2020. https://web.archive.org/web/20200802084355/https://www.bostonglobe.com/.

[21] The Daily Caller: Internet archive's copy from September 5, 2020. https://web.archive.org/web/20200905133311/https://dailycaller.com/.

[22] The WARC format 1.0. https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.0/.

[23] WARC revisit tag. https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.0/#revisit.

[24] Wayback machine. https://www.archive.org/web.

[25] Webpack. https://webpack.js.org/guides/tree-shaking/.

[26] Webrecorder. https://webrecorder.net/.

[27] S. G. Ainsworth, A. Alsum, H. SalahEldeen, M. C. Weigle, and M. L. Nelson. How much of the web is archived? In *Joint Conference on Digital Libraries*, 2011.

[28] A. AlSum, M. C. Weigle, M. L. Nelson, and H. Van de Sompel. Profiling web archive coverage for top-level domain and content language. *International Journal on Digital Libraries*, 2014.

[29] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *DSN*, 2011.

[30] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *ICSE*, 2011.

[31] J. F. Brunelle, M. Kelly, H. SalahEldeen, M. C. Weigle, and M. L. Nelson. Not all mementos are created equal: Measuring the impact of missing resources. *International Journal on Digital Libraries*, 2015.

[32] J. F. Brunelle, M. Kelly, M. C. Weigle, and M. L. Nelson. The impact of javascript on archivability. *International Journal on Digital Libraries*, 2016.

[33] J. F. Brunelle, M. C. Weigle, and M. L. Nelson. Archival crawlers and javascript: discover more stuff but crawl more slowly. In *Joint Conference on Digital Libraries*. IEEE, 2017.

[34] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *NSDI*, 2015.

[35] Z. T. Fernando, I. Marenzi, and W. Nejdl. ArchiveWeb: Collaboratively extending and exploring web archive collections—how would you like to work with your collections? *International Journal on Digital Libraries*, 2018.

[36] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener. A large-scale study of the evolution of web pages. *Software: Practice and Experience*, 2004.

[37] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[38] A. Goel, V. Ruamviboonsuk, R. Netravali, and H. V. Madhyastha. Rethinking client-side caching for the mobile web. In *HotMobile*, 2021.

[39] H. Holzmann, V. Goel, and A. Anand. Archivespark: Efficient web archive access, extraction and derivation. In *Joint Conference on Digital Libraries*, 2016.

[40] International Internet Preservation Consortium. Access Working Group. Use cases for access to internet archives. *IIPC Report*, 2006.

[41] M. Kelly, M. L. Nelson, and M. C. Weigle. A framework for aggregating private and public web archives. In *Joint Conference on Digital Libraries*, 2018.

[42] R. Ko, J. Mickens, B. Loring, and R. Netravali. Oblique: Accelerating page loads using symbolic execution. In *NSDI*, 2021.

[43] J.-w. Kwon and S.-M. Moon. Web application migration with closure reconstruction. In *WWW*, 2017.

[44] S. Lawrence, F. Coetzee, E. Glover, G. Flake, D. Pennock, B. Krovetz, F. Nielsen, A. Kruger, and L. Giles. Persistence of information on the web: Analyzing cita-

tions contained in research articles. In *CIKM*, 2000.
[45] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 1966.
[46] T. Libert. Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites. *International Journal of Communication*, 2015.
[47] J. Lin, M. Gholami, and J. Rao. Infrastructure for supporting exploration and discovery in web archives. In *WWW*, 2014.
[48] B. Loring, D. Mitchell, and J. Kinder. ExpoSE: Practical symbolic execution of standalone JavaScript. In *SPIN Symposium on Model Checking of Software*, 2017.
[49] S. Mardani, A. Goel, R. Ko, H. V. Madhyastha, and R. Netravali. Horcrux: Automatic javascript parallelism for resource-efficient web computation. In *OSDI*, 2021.
[50] C. C. Marshall and F. M. Shipman. On the institutional archiving of social media. In *Joint Conference on Digital Libraries*, 2012.
[51] J. Mickens. Rivet: Browser-agnostic remote debugging for web applications. In *USENIX ATC*, 2012.
[52] J. W. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, 2010.
[53] J. Nejati, M. Luo, N. Nikiforakis, and A. Balasubramanian. Need for mobile speed: A historical analysis of mobile web performance. In *TMA*, 2020.
[54] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, 2016.
[55] R. Netravali and J. Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *NSDI*, 2018.
[56] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring time-to-interactivity for web pages. In *NSDI*, 2018.
[57] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX ATC*, 2015.
[58] A. Ntoulas, J. Cho, and C. Olston. What's new on the web? the evolution of the web from a search engine perspective. In *WWW*, 2004.
[59] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, 2010.
[60] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *FSE*, 2013.
[61] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.
[62] D. Spinellis. The decay and failures of web references. *Communications of the ACM*, 46(1):71–77, 2003.
[63] O. Tripp and O. Weisman. Hybrid analysis for javascript security assessment. In *ESEC/FSE*, 2011.
[64] X. Wang and Z. Xie. The case for alternative web archival formats to expedite the data-to-insight cycle. In *Joint Conference on Digital Libraries*, 2020.
[65] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *NSDI*, 2013.
[66] S. Wei and B. G. Ryder. Practical blended taint analysis for javascript. In *International Symposium on Software Testing and Analysis*, 2013.

A Artifact Appendix

A.1 Abstract

Our open-source artifact contains the scripts and the data necessary to produce the key results from this paper. It also contains the code for the analysis framework which informed Jawa's design.

A.2 Scope

The artifact can be used to confirm the three main benefits of Jawa: a) reduced storage overhead, b) improved fidelity by eliminating almost all failed network requests, and c) improved crawling throughput.

A.3 Contents

The artifact contains all the code required to generate the key results with respect to three metrics: storage, fidelity and throughput. This includes a) Jawa's filter list and a NodeJS based crawler that leverages this filter list while loading web pages; b) a NodeJS based analyzer that injests JS files and instruments them to track all the JS functions executed at runtime, the set of event handlers registered, and the return values of browser APIs; and c) a set of scripts to automatically run the above code on a given corpus of pages. These scripts will produce the following results:

- **E1**: Reduced storage overhead using Jawa's two techniques: eliminating non-functional code using the filter list, and eliminating unused code by tracking the set of functions executed during the page load plus those required for enabling user interactions. This result will mimic the trend shown in Figure 7.

- **E2**: Improved page fidelity by eliminating almost all failed network requests. This result will reproduce the number of failed requests and the corresponding number of bytes not fetched, as shown in Figure 8.

- **E2**: Improved crawling throughput by reducing the number of IOs on the crawling index. This result will mimic the trend shown in the *"Crawling"* column of Table 3.

Apart from the scripts, the artifact contains a corpus of 3000 pages which is pre-recorded using the Mahimahi [57] tool. All scripts are run on this corpus of pages. Finally, the artifact also contains the JS analysis framework which was used to inform Jawa's design choices (§3).

A.4 Hosting

The source code of the artifact is hosted on https://github.com/goelayu/Jawa with the corresponding commit ID: *"07e358eeed7cc054747271b19070b5563f3ff189"*. The corpus of pages is hosted on Google Drive.

A.5 Requirements

Software dependencies

The artifact has been tested on Ubuntu 16.04.7 LTS. It requires installing the following dependencies, in addition to the NodeJS dependencies included in the github repo (§A.6):

```
$ sudo apt-get install mahimahi google-chrome-
    stable parallel r-base r-base-core
$ sudo sysctl -w net.ipv4.ip_forward=1
```

A.6 Installations

Setting up the artifact involves three steps: a) downloading the source code and installing the NodeJS dependencies, b) patching the NodeJS dependencies to use the modified versions included in the github repo, and c) fetching and extracting the corpus of pages to run the analysis on.

Install the code

```
$ git clone https://github.com/goelayu/Jawa
$ cd Jawa
$ npm install
$ export NODE_PATH=${PWD}
```

Patch the dependencies

```
$ vim node_modules/puppeteer-extra-plugin-
    adblocker/dist/index.cjs.js
# add to line 73:
  return adblockerPuppeteer.PuppeteerBlocker.parse
    (fs.readFileSync('../filter-lists/combined-
    alexa-3k.txt', 'utf-8'));
```

Fetch the data

```
$ cd data
# download tarball from https://drive.google.com/
    file/d/17j6AYgaaXMhmV0VKWUmU_kMcHibMryVV/view?
    usp=sharing
$ tar -xf corpus.tar
```

A.7 Experiments workflow

As listed in §A.3, the artifact scripts will produce results corresponding to three metrics: storage, fidelity and crawling throughput.

A.7.1 Fidelity

We provide scripts and data to exactly reproduce Figure 8 (both a and b). The corpus used for this experiment contained 3000 pages. On a single core machine, it takes roughly 20–30 seconds for each page to load and, therefore, takes about 20 hours to load all 3000 pages once. We recommend to either run this experiment on a smaller corpus of pages (more details below) or to use a multi-core (16–32 cores) machine to speed up the overall execution time.

```
$ cd ../ae
# Usage: ./fidelity.sh <corpus_size> <num of
    parallel processes>
$ ./fidelity.sh 3000 1 # depending on the number
    of available cores on your machine, provide
    the 2nd argument
```

The output graphs will be generated in the same directory: *"count_fidelity.pdf"* and *"size_fidelity.pdf"*, corresponding to Figures 8(a) and 8(b), respectively.

A.7.2 Storage

Reproducing Figure 7 requires processing 1 million pages, which would take around a week (even with 128 CPU cores). We instead provide scripts to process 3000 pages, and demonstrate storage savings derived from both of Jawa's techniques. We provide preprocessed web pages, i.e., injected with instrumentation code to detect which functions are executed at runtime, and code to track event handlers. You can fetch the the instrumented pages as follows:

```
$ cd ../data
# download tarball from https://drive.google.com/
    file/d/16Pt4a2l1CNxC8UBwjalgEki-UlGAnFUm/view?
    usp=sharing
$ tar -xf processed.tar
```

You can now run the end-to-end storage analysis script:

```
$ cd ../ae
# Usage: ./storage.sh <corpus_size> <num of
    parallel processes>
$ ./storage.sh 3000 1 # depending on the number of
     available cores on your machine, provide the
    2nd argument
```

The above script will print three storage numbers (in bytes) to the console. a) Total JS storage after deduplication (as incurred by Internet Archive); this mimics the *"IA*"* bar in Figure 7(a). b) Total JS storage after applying Jawa's filter; this mimics the *"IA*+Combined Filter"* bar in Figure 7(a). c) Total JS storage after removing unused JS functions; this mimics the *"Jawa"* bar in Figure 7(a).

A.7.3 Crawling throughput

We reproduce the throughput results from Table 3's *"Crawling"* column. The storage script above outputs the crawling index IOs as well. It prints the following two numbers: a) reductions in crawling IOs for the 50th percentile page, and b) reductions in crawling IOs for the 95th percentile page.

Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update

Chijun Sima* *Tencent*
Yao Fu* *The University of Edinburgh*
Man-Kit Sit *The University of Edinburgh*
Liyi Guo *Tencent*
Xuri Gong *Tencent*
Feng Lin *Tencent*
Junyu Wu *Tencent*
Yongsheng Li *Tencent*
Haidong Rong *Tencent*
Pierre-Louis Aublin *IIJ research laboratory*
Luo Mai *The University of Edinburgh*

Abstract

Deep Learning Recommender Systems (DLRSs) need to update models at low latency, thus promptly serving new users and content. Existing DLRSs, however, fail to do so. They train/validate models offline and broadcast entire models to global inference clusters. They thus incur significant model update latency (e.g. dozens of minutes), which adversely affects Service-Level Objectives (SLOs).

This paper describes Ekko, a novel DLRS that enables low-latency model updates. Its design idea is to allow model updates to be immediately disseminated to all inference clusters, thus bypassing long-latency model checkpoint, validation and broadcast. To realise this idea, we first design an *efficient peer-to-peer model update dissemination algorithm*. This algorithm exploits the sparsity and temporal locality in updating DLRS models to improve the throughput and latency of updating models. Further, Ekko has a *model update scheduler* that can prioritise, over busy networks, the sending of model updates that can largely affect SLOs. Finally, Ekko has an *inference model state manager* which monitors the SLOs of inference models and rollbacks the models if SLO-detrimental biased updates are detected. Evaluation results show that Ekko is orders of magnitude faster than state-of-the-art DLRS systems. Ekko has been deployed in production for more than one year, serves over a billion users daily and reduces the model update latency compared to state-of-the-art systems from dozens of minutes to 2.4 seconds.

1 Introduction

Deep Learning Recommender Systems (DLRSs) are a key infrastructure in large technology organisations such as Meta [54], ByteDance [23], Google [15] and NVIDIA [56]. A DLRS often contains a large group of *parameter servers* that host numerous Machine Learning (ML) models (i.e. embedding tables [10, 26, 54] and deep neural networks [18]). The parameter servers are replicated in geo-distributed data centres for fault-tolerance and low-latency communication with clients. Each data centre has a group of *inference servers* which pull models from local parameter servers and serve clients with recommendation results. To ensure new users and content can be served promptly, a DLRS must update ML models continuously: it first uses *training servers* to collect new training data and compute model gradients. It then uses parameter servers to disseminate model updates to model replicas, usually through a Wide-Area Network (WAN).

Large-scale DLRSs need to serve billions of users [15, 23, 54] and they must achieve latency-related Service-Level Objectives (SLOs) [49], e.g. the latency of making a newly created content available to users. To best achieve SLOs, the operators of DLRSs have emerging requirements for achieving low latency in updating models. There are several reasons for this: (i) recent DLRS applications (e.g. YouTube [24] or TikTok [8]) have enabled users to create massive short videos, articles and images. All these contents need to be made available for clients as soon as possible, usually in minutes if not seconds; (ii) data protection laws (e.g. GDPR [60]) allow DLRS users to become anonymous. The behaviours of anonymous users need to be learnt online; (iii) numerous online ML models (e.g. reinforcement learning [74]) have been adopted in production to improve recommendation quality. These models must be continuously updated online to achieve the best possible performance.

Unfortunately, achieving low-latency model updates is extremely difficult in existing DLRSs. Existing systems such as Merlin [56], TFRA [66], Check-N-Run [21] and BigGraph [39] follow an *offline* approach to updating models: after having collected new training data, these systems compute gradients for models offline, validate model checkpoints, and broadcast the checkpoints to all data centres. Such a model update process can take minutes and even hours [21]. An alternative approach is to use WAN-optimised ML systems [28] or federated learning systems [37]. These systems update replicated models using locally collected data and lazily synchronise replicas. The lazy synchronisation, however, introduces a non-trivial level of asynchrony, which often

*Chijun and Yao are co-primary authors.

adversely affects the achievement of SLOs [28, 42].

We want to explore a DLRS design that can achieve low-latency model updates without compromising SLOs. Our key idea is to allow training servers to update models (using gradients) online and immediately disseminate model updates to all inference clusters. This design allows us to bypass long-latency update steps, including offline training, model checkpoint, validation and broadcast, thereby reducing model update latency. To make this design feasible, we need to address several challenges: (i) how to efficiently disseminate massive model updates over WANs which have limited bandwidths and heterogeneous network paths [28]; (ii) how to protect SLOs from network congestion that can delay critical updates; and (iii) how to protect SLOs from biased model updates that are detrimental to model accuracy.

This paper introduces Ekko, a novel large-scale DLRS that updates globally replicated models at low latency. The design of Ekko makes several key contributions:

(1) Efficient peer-to-peer model updates dissemination. Existing parameter servers often adopt primary-backup data replication protocols [11, 41, 67] to realise model updates. With massive model updates, however, primary-backup protocols exhibit insufficient scalability due to long update latency [67] and leader bottlenecks [2].

To address these issues, we explore how to enable Peer-to-Peer (P2P) [20] model update dissemination. We design an efficient *log-less state-based synchronisation algorithm* for geo-distributed DLRSs (see §4). This algorithm is effective in DLRSs because model updates often hit hot parameters [21], and it only transfers the latest version of a model parameter (i.e. state). Ekko must allow parameter servers to efficiently discover the differences of model states in a P2P manner. To this end, we design (i) *model update caches* that allow parameter servers to efficiently track and compare model states, (ii) *shard versions* that can significantly reduce network bandwidth consumption when comparing model states, and (iii) *WAN-optimised dissemination topologies* that allow parameter servers to prioritise bandwidth-affluent intra-DC network paths over bandwidth-limited inter-DC network paths.

(2) SLO protection mechanisms. Ekko allows model updates to reach inference clusters without offline model validation. Such a design can make SLOs (particularly those related to the freshness and quality of recommendation results) vulnerable to network congestion and biased updates, both possible in production environments.

To handle network congestion, we design an *SLO-aware model update scheduler* (see §5). This scheduler computes metrics, including the update freshness priority, the update significance priority and the model priority. These metrics predict the impact of model updates on the inference SLOs. The scheduler computes a priority for each model update online based on these metrics. We integrate the scheduler into parameter servers without changing the decentralised architecture of the P2P model update dissemination in Ekko.

Ekko handles biased updates using a novel *inference model state manager*. This manager creates a *baseline* model for each group of inference models. This baseline model receives a small amount of user traffic and serves as the ground truth to the inference model. The manager continuously monitors the quality-related SLOs for baseline and inference models. When biased model updates corrupt the state of the inference model, the manager notifies *witness servers* to roll back the model to a healthy state.

We evaluate Ekko using both test-bed and large-scale production clusters (see §6). Test-bed experimental results show that Ekko reduces the model update latency by up to 7× compared to state-of-the-art parameter servers, namely Adam [11]. We further run large-scale production experiments with 40 TB models and over 4,600 servers spread across geo-distributed regions. Experimental results show that Ekko disseminates updates in 2.4 seconds while executing 1 billion updates per second (i.e. 212 GB/s). Ekko only uses 3.0% of the total network bandwidth for synchronisation, leaving the rest for training and inference. This second-level latency performance is orders of magnitude faster than the minute-level latency (i.e. 5 minutes [69]) achieved by state-of-the-art DLRS infrastructures (e.g. TFRA [66] and Check-N-Run [21]).

2 Low-Latency Model Updates in DLRSs

In this section, we introduce DLRSs and their algorithms for updating models. We then describe their Service-Level Objectives (SLOs) that can benefit from reducing the latency of updating models. Finally, we discuss the system challenges associated with realising low-latency model updates.

2.1 DLRSs and model updates

Most technology organisations adopt DLRSs following a system architecture shown in Figure 1. A DLRS often serves clients distributed across the globe (❶). To minimise serving latency, DLRS models (i.e. embedding tables [10, 26, 54] and deep neural networks [18]) are geo-replicated in multiple data centres. When a client's request arrives, an *inference server* pulls the model parameters from local parameter servers and infers over this model to answer the request.

Data pipelines collect training data (e.g. new content and user activities) from clients at run-time. The collected data reach training servers in a data centre (❷). The training servers use optimisers [33] to compute gradients that correct corresponding models. All updated models (usually 100s - 1,000s) are persisted as checkpoints (❸). The checkpoints are first validated, and only those that can improve SLOs are disseminated to the parameter servers in inference-oriented data centres over a WAN (❹), finishing the model update process.

In practice, the latency of updating a DLRS model comprises the time of computing model updates and disseminat-

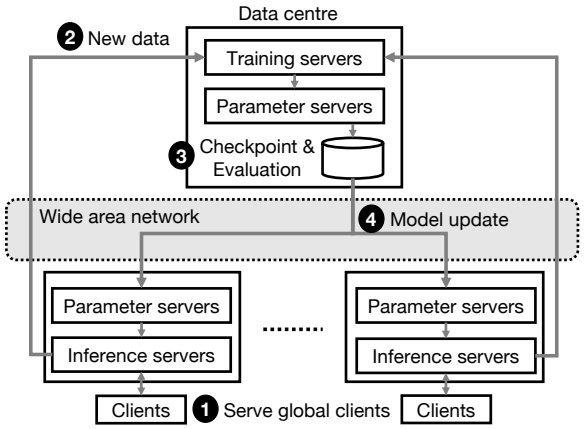


Figure 1: A typical DLRS architecture.

ing the updates to global data centres. This latency definition presumes that we have used low-latency message queues, e.g. Kafka [36], to accelerate the training data ingestion. Recent DLRSs, e.g. NVIDIA Merlin [56] and Meta Check-N-Run [21], report minute-level and hour-level latencies in updating models. Suppose we want to update a DLRS model with a large embedding table (often several TB in size). In this case, it can take tens of minutes to persist this model as a checkpoint and validate the model. It takes another dozen minutes to disseminate this model over a WAN (assuming this WAN provides several Gbps bandwidth [72]).

2.2 Reasons for low-latency model updates

DLRSs need to achieve numerous SLOs (usually related to the freshness and quality of recommendation results). Take a short-video recommendation service (e.g. TikTok) as an example. The DLRS model accuracy determines this service's quality SLOs. The time of making freshly made videos accessible to users decides this service's freshness SLOs.

In real-world DLRSs, we observe that SLOs often depend on the latency of finishing model updates, making low-latency model updates a critical system requirement. There are several reasons for this:

(1) Massive new content created in a short time. Global DLRSs, e.g. YouTube [24], TikTok [8] and Instagram [22], often serve billions of users, and they allow users to create massive content quickly. The DLRSs need to quickly incorporate the created content into recommendation results — by updating their models at low latency — otherwise affecting user engagement.

(2) Increasing anonymous users. Data protection laws (e.g. GDPR [60]) have forbidden many DLRSs from tracking user activities. As a result, such a DLRS can have anonymous users yet unknown to the recommendation models, even though these users have used the same service before. A DLRS thus must quickly react to the online activities of anonymous users, thus meeting their recommendation requirements. Such a quick reaction depends on low-latency model updates.

(3) Increasing online recommendation models. DLRSs have increasing online ML models, e.g. those using reinforcement learning [74] and continual learning [69]. These models improve recommendation quality. They need to collect training data from online user activities, and they thus must continuously update model parameters at low latency.

2.3 Our key idea and associated challenges

We want to explore how to achieve low latency in updating DLRS models. Our observation is that the update latency is accumulated mainly due to several offline steps: model training, validation and broadcast. Suppose we bypass these offline steps and allow updated models to be disseminated to the inference clusters directly. In that case, we can vastly reduce the steps for updating models, thus achieving low latency. To realise such a design, however, we must address several challenges:

(1) Lack of efficient algorithms for disseminating massive model updates. A real-world DLRS often has a large number of models (e.g. usually 100s - 1,000s). It needs to update many of these models online. These models comprise those on a multi-stage recommendation pipeline [10, 15] and those for A/B tests [69]. These models often cost 10s of TB memory. They have the requirement to complete massive model updates online (e.g. 100s of GB per second).

Suppose we use conventional data replication protocols, e.g. chain replication [41] and two-phase commit [11]. These protocols target generic data replication. They lack mechanisms to coordinate ML model updates (which may exhibit different impacts on inference SLOs) over a bandwidth-limited network (i.e. WAN). Furthermore, these conventional protocols suffer from leader bottlenecks. They also incur long update latency caused by the heterogeneous WAN paths and network stragglers. As a result, these protocols are ill-suited to meet our high-throughput, low-latency requirements. Alternatively, we could use geo-replication protocols [72]. These protocols, however, cannot handle the failures of servers in the training data centres, making them unable to meet our system availability requirement.

We also considered network-efficient distributed ML systems, e.g. Gaia [28] and Google Federated [35]. These systems [7, 28, 35, 37, 46] allow models to be trained independently in each data centre, thus improving the throughput and latency of updating models. They, however, lazily synchronise their states and therefore incur stale model states [47], which can adversely affect recommendation quality. As a result, the loosely synchronised distributed ML systems cannot meet our model accuracy requirement.

(2) Lack of mechanisms for protecting SLOs. Enabling online model updates in a DLRS poses challenges to SLOs. Such a DLRS can have model updates competing for network

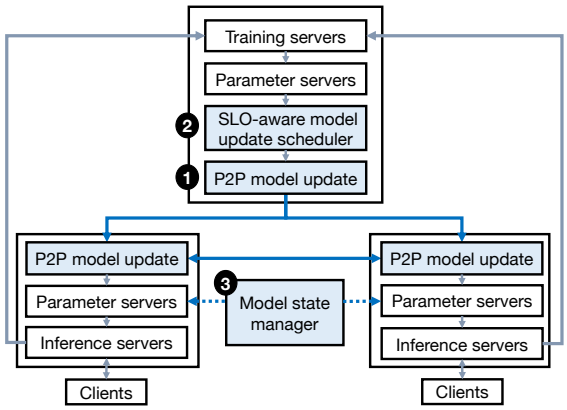


Figure 2: Ekko architecture overview.

bandwidth, delaying critical updates (e.g. those that significantly affect model accuracy or bring new items online). Even though there are systems that schedule the sending of model gradients [6], these systems target training clusters. As a result, they prioritise model updates based on gradients [6, 28] and lack awareness of how those updates will affect the SLOs of inference models.

Online model updates can be even detrimental. Since online updates are often computed based on a small batch of data (collected in a short time window: seconds or minutes), they often contain noise [34]. When updates become particularly noisy, they become detrimental to inference SLOs (i.e. decrease the accuracy of inference models). To handle this, existing model serving systems, e.g. Clipper [16] and Clockwork [25], use offline model validation, which averages model updates accumulated for an extended period (e.g. hours). Other model serving systems, e.g. Google TFRA [66], track the SLO metrics of inference models, and they reload checkpoints when SLOs are deteriorating. Such a design, however, is challenging to implement in DLRSs. Giant DLRS models (e.g. recommendation-oriented transformers [18]) are increasingly common. Reloading these models affect the availability of services.

3 Ekko System Architecture

This paper introduces Ekko, a novel DLRS system that enables low-latency model updates. In this section, we describe the system model of Ekko and present an overview that highlights the novel components in Ekko.

3.1 System model

Ekko is a geo-distributed DLRS. It updates models in a central data centre. It then disseminates updated models to geo-distributed data centres close to global users (i.e. clients). Ekko represents models as key-value pairs, and it partitions the models into shards (e.g. 100,000 in our production environment). It stores model shards in key-value stores (named as a parameter store in Ekko). The parameter stores assign key-value pairs to shards through hashing. The model size can change over time since the model often incorporates new items and feature expiration online [32].

Ekko directs parameter requests to model shards using software-based *routers*. The routers designate parameter servers in the training DC as the primaries for model shards. They also ensure that the choice of primaries can balance the workload of parameter requests. The implementation of the routers follows typical key-value stores and databases [38]. We omit the details of the router implementation in this paper.

In the routers, there are shard managers which can handle resource overload, fault domains [55] and copyset issues [12]. Different from conventional shard managers, Ekko's shard managers realise several DLRS-specific optimisations: (i) To amortise request processing overhead, Ekko batches concurrent inference requests for the same model [16]. Batched requests, however, can query a large number (e.g. 1000s) of parameters on different parameter servers, resulting in long-tail query latency [19]. To prevent long-tail latency, Ekko limits the number of servers assigned to a model's shards; (ii) Ekko supports multiple DLRS applications which require performance isolation. It maps the shards of different applications to different servers. Therefore, the spike of requesting the shards of an application will not affect the shards of other applications.

3.2 Architecture overview

We highlight the novel designs in Ekko in Figure 2. As we can see, Ekko enables parameter servers to achieve efficient *peer-to-peer model updates* (❶) (see §4). The P2P model update algorithm prevents the central training data centre from broadcasting updated models. Instead, it uses all network paths inside and across data centres (those solid lines in the figure), thus achieving high throughput in disseminating model updates. Without using a central coordinator, each data centre can independently choose optimised intervals that synchronise model updates.

Ekko supports concurrent dissemination of massive model updates. These updates can compete for network resources, delaying the updates that largely benefit SLOs. To handle this, Ekko relies on *an SLO-aware model update scheduler* (❷) (see §5.2). This scheduler predicts how each model update will affect inference results. The prediction results facilitate the computation of the priority of each model update. Based on the priority, Ekko coordinates which model updates to disseminate first at the training data centre, thus improving the overall satisfaction of the SLOs on inference servers.

Ekko can protect inference servers from being affected by detrimental model updates. To achieve this, it has a *model state manager* (❸) (see §5.3) running in the inference clusters.

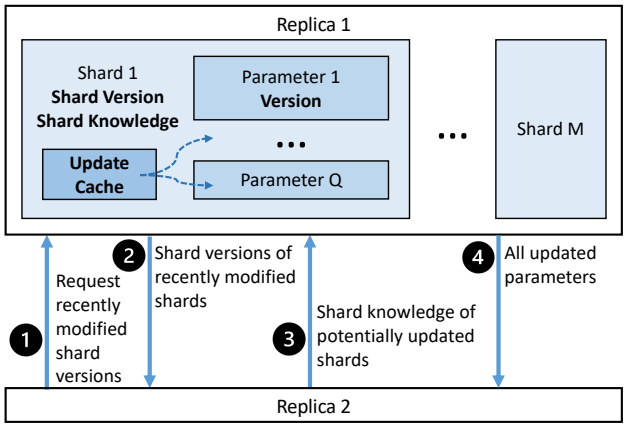


Figure 3: Ekko P2P model update overview.

This model state manager monitors SLO-related metrics of inference models. Suppose an inference model shows downgraded performance (caused by online updates). In that case, the manager rollbacks the model's state to a better-performing one, thus recovering the performance of the inference model.

4 Efficient Peer-to-Peer Model Update

This section introduces the efficient P2P model update mechanism in Ekko. To enable P2P model update in parameter servers, the design of Ekko achieves the following goals:

- Ekko needs to coordinate a large number (e.g. thousands) of parameter servers (deployed across the globe) to finish model updates. To avoid stragglers (which can be caused by slow networks), we design log-less synchronisation for the parameter servers in Ekko (§4.3).
- As a shared DLRS, Ekko needs to host thousands of models. These models can generate massive (e.g. billions per second) updates online. To support this, Ekko enables parameter servers to efficiently discover model updates through peers and pull updates without using excessive computation and network resources (§4.4).
- Ekko needs to support geo-distributed deployments, which often involve heterogeneous network paths across WANs and server/network failures. To support this, Ekko has system designs that improve the throughput/latency of sending model updates over a WAN and tolerate server/network failures (§4.5).

In the following, we give an overview of the P2P model update mechanism and describe its implementation in detail.

4.1 Model update overview

Figure 3 highlights the components and steps involved in a model update in Ekko. Suppose that we want to synchronise a shard (denoted by shard 1) between two replicas (denoted by replica 1 and replica 2). Similar to all other shards, shard 1 has a (i) *shard knowledge* which summarises parameter updates, and (ii) an *update cache* that tracks recent model updates based on *parameter versions*. Each shard also associates a *shard version* which tells if this shard potentially has parameters to synchronise. The shard knowledge, update cache and shard version together accelerate parameter synchronisation among parameter servers.

To finish a model update, replica 2 requests the recently modified shard versions from replica 1 (❶). Once receiving the request, replica 1 returns a list of recently modified shard versions (❷). Replica 2 then compares all shard versions of replica 1 with its local shard versions and then sends related shard knowledge to replica 1 (❸). Finally, replica 1 sends all updated parameters to replica 2 (❹). Following these steps, Ekko can ensure that model updates are eventually disseminated to all replicas at low latency (i.e. eventual consistency).

We find eventual consistency acceptable in real-world DLRSs. Even though DNN replicas may diverge in a small time window, they often exhibit close (even often identical) inference results [11]. This is because DNNs often use floating-point numbers to represent model parameters, and therefore, DNN replicas make close predictions even though there is a slight difference in the values of their local parameters.

4.2 Parameter versions in DLRSs

To track the state of model parameters, Ekko assigns each key-value pair (i.e. the storage format of a model parameter) with a parameter version defined below:

Definition 1 (Parameter Version)**.** A Parameter Version v is a pair (t, id) that consists of a timestamp t and an id uniquely identifying a replica. The timestamp t is generated based on the time range provided by modern physical time sources [14, 43]. Ekko makes sure t increases monotonically in each replica and pads the physical timestamp with a counter to make sure any two updates that originate from a single replica do not share the same timestamp. We define the total order of Parameter Versions:

$$v_1 \geq v_2 \iff (t_1 > t_2) \mid ((t_1 = t_2) \wedge (id_1 \geq id_2))$$

A parameter with a larger Parameter Version supersedes another during conflict resolution [62].

In Ekko, it is worth noting that the timestamp is based on a real-time clock instead of a logical clock (which is often used in key-value stores and storage services). We find such a design effective in distributed DLRSs for a reason: a DLRS has embedding tables where parameters are sparsely updated. Suppose there is an embedding's parameter in a primary replica and this parameter has a significant update count, but the primary does not disseminate this parameter

before it fails. When this primary recovers, the counter can overwrite the current primary with a small update count. Such an overwrite can adversely affect recommendation quality because the overwritten primary can have a newer parameter (updated by recently collected training data), leading to better recommendation results. Hence a logical counter is not sufficient to resolve conflicts in distributed DLRSs.

4.3 Log-less parameter synchronisation

Once version numbers have been assigned to parameters, Ekko needs to decide how to synchronise the different replicas. We observe that a DLRS often overwrites parameters and only the last write decides the state of a parameter. We therefore decide to send the last version of parameters.

Ekko needs to decide the interval of synchronising replicas. We could use log-based synchronisation algorithms [9, 11]: these algorithms choose synchronisation intervals so that model updates can be sent at the rates that do not exceed the bandwidth on the *slowest* links in a network. These algorithms, however, cause the under-utilisation of many network links. More importantly, it results in stragglers which can significantly increase the latency of synchronisation, making parameter servers more likely to have stale states when they recover from failures. Hence, we want to realise log-less parameter synchronisation in parameter servers so that these servers can dynamically choose synchronisation intervals with their peers according to the bandwidth on each link.

Shard knowledge in parameter servers. We propose to use shard knowledge [50, 51] to realise log-less parameter synchronisation. More formally, in each replica, all its shards maintain a corresponding shard knowledge. The shard knowledge, implemented using version vectors [58], summarises the parameter updates they have learnt. Shard data (associated with the shard knowledge VV_{shard}) reflect the state of an empty shard applying all historical parameter updates originating from each replica r, where the update corresponding parameter version $v \leq VV_{shard}[r]$. Suppose there is an update for the parameter p to be processed in replica r. To maintain shard knowledge, this replica generates a new parameter version $v_p = (t, id)$ and sets $VV_{shard}[id] = v_p$.

Shard synchronisation process. To synchronise a shard, replica r sends its shard knowledge VV_{r_1} to a selected replica s. Replica s *records* its current shard knowledge VV_s — that is, it atomically reads out VV_s and selects from its store all parameters p whose parameter version $v_p = (t_p, id_p) > VV_{r_1}[id_p]$ — and responds to r with VV_s. Then, r atomically applies all parameter updates based on the response from s, and further merges VV_s with its current shard knowledge VV_{r_2}.

There are several considerations to note in the synchronisation process: (i) When replica r synchronises with replica s, r could have concurrent synchronisation operations with another replica (denoted as replica k). These operations can complete before r finishes processing the response from s. As a result, VV_{r_2} (which is the result of $VV_r \bigsqcup VV_k$) does not necessarily equal VV_{r_1}. (ii) The synchronisation process omits all superseded versions of an updated parameter in failure-free scenarios where the requests for updating a parameter are always routed to the same primary. We find these failure-free scenarios common in our production environments.

4.4 Making synchronisation efficient

Ekko must ensure parameter synchronisation have negligible performance overheads on parameter servers. Otherwise, synchronisation can consume excessive computation and communication resources, affecting parameter servers' performance in serving model inference and training requests. In the following, we discuss how to make parameter synchronisation efficient through parameter update caches (which reduce computation costs) and shard versions (which reduce communication costs).

4.4.1 Parameter update caches

Since a shard can have a large number of parameters, naively iterating all parameters to answer a synchronisation request incurs substantial computation costs. Even though we could use an index to accelerate the parameter iteration, maintaining such an index costs tremendous memory resources, which are difficult to provision on parameter servers.

We design parameter update caches to reduce the computation cost of parameter synchronisation. The design of such caches exploits the *sparsity* and *temporal locality* we often observe in DLRSs [21]. Unlike dense DNN training systems where the entire models are updated every iteration, a DLRS updates a subset of its parameters (i.e. sparsity). For example, in our production DLRSs, 3.08% of its parameters are updated per hour. Further, model updates are often overwriting certain parameters (i.e. temporal locality) in a time window. This is because a DLRS often has trendy items and users, and their parameter updates dominate in a short period.

More specifically, a parameter update cache contains pointers to recently updated parameters. It exploits a Dominator Version Vector (denoted as DVV) to judge whether to hit the cache when a synchronisation request arrives.

Cache maintenance algorithm. The maintenance of the cache guarantees two invariants: (i) for all parameters $p_{uncached}$ existing in a shard but not in the cache, $DVV[id_{p_{uncached}}] \geq v_{p_{uncached}}$; (ii) for all cached parameters p_{cached}, $DVV[id_{p_{cached}}] < v_{p_{cached}}$.

Algorithm 1 describes the maintenance of the parameter update cache in Ekko. The maintenance relies on the estimated update propagation time D_{prop}. Consider the function of updating the cache: `UpdateCache` (line 1). $t_{pruneto}$ is a timestamp that describes $DVV_{proposed}$ – a version vector that judges whether a parameter should be pruned. For every modification request, the cache records a pointer to that parameter

Algorithm 1: Update Cache Maintenance using D_{prop}

```
1  Function UpdateCache(p):
2      if v_p.t ≤ t_pruneto then
3          DVV.Merge(v_p);
4      else
5          cache.Add(p);
6      end
7  Function PruneCache():
8      t_pruneto ← max(t_pruneto, t_now − D_prop);
9      for p ∈ cache do
10         if v_p.t ≤ t_pruneto then
11             cache.Erase(p);
12             DVV.Merge(v_p);
13         end
14     end
```

if the parameter version $v_p = (t_p, id_p)$ of the modified parameter p is larger than $DVV_{proposed}[id_p]$ (line 5). Otherwise, the cache merges the parameter version with DVV (line 3).

Consider the function of pruning a parameter pointer: `PruneCache` (line 7). This function takes D_{prop}, which essentially allows Ekko to exploit online observations towards cache hit rates to guide cache pruning operations. Suppose we want to prune parameter pointers when the cache size has grown beyond a limit. In that case, the cache first determines $DVV'_{proposed}$, which strictly dominates $DVV_{proposed}$ (line 8). It then removes parameter pointers dominated by $DVV'_{proposed}$ (line 11). Eventually, the cache updates DVV by merging it with parameter versions of pruned parameters (line 12). By doing so, Ekko achieves adaptive management of the cache size, reducing its memory footprint.

Cache hit analysis. We analyse when parameter updates hit the cache. Suppose replica s receives the synchronisation request from replica r which holds the shard knowledge VV_r. If VV_r dominates DVV_s, the request hits the cache and its subsequent operations (e.g. selecting a parameter) only touch the parameters in the cache.

Ekko ensures that the use of the update cache does not affect the eventual consistency property of log-less parameter synchronisation: the synchronisation process needs to select out parameters p in s where $v_p > VV_r[id_p]$. Because the update cache holds the invariant that $DVV_s[id_{p_{uncached}}] \geq v_{p_{uncached}}$ and VV_r dominates DVV_s, the process selects out the same set of parameters as the previous algorithm.

The parameter update caches are particularly effective in reducing the cost of selecting parameters. According to the traces of the caches deployed in our production environments, 99.4% of the synchronisation requests can hit the caches, leading to a 99% reduction in the cost of selecting parameters.

4.4.2 Shard versions

We introduce *shard versions* to reduce network costs in synchronising replicas. Shard versions capture partial causality relationships of shard data on replicas, and they are much smaller than version vectors. We can allow the replicas to book-keep shard version lists where each list is associated with a neighbour replica. By doing this, replicas can identify potentially updated shards by exchanging and comparing shard version lists. Formally, we define shard versions as:

Definition 2 (Shard Version)**.** A *shard version* $sv = (c, id)$ is a pair consisting of a counter c, which is monotonically incremented in each shard of each replica, and an id identifying the replica that generates this version. $sv_1 \succeq sv_2$ of a same shard s if and only if $id_1 = id_2$ and $c_1 \geq c_2$.

Shard version maintenance. On initialisation, each replica generates shard versions for its shards. It later generates a new shard version when a training worker issues a parameter update. Since each shard has a primary replica, there is a single replica generating shard versions in normal cases.

Once receiving a synchronisation request, the responder replica, denoted as s, replies its shard version: sv_s together with VV_s and updated parameters. Once having this reply, the requester replica, denoted as r, finishes the following operations in an atomic manner: it (1) merges its shard knowledge VV_r with the received VV_s (The merging result is denoted as VV'_r), and (2) it updates its shard version sv'_r to be sv_s when $VV'_r = VV_s$; Otherwise, replica r generates a new shard version if $VV'_r \neq VV_r$. Note that: when VV_r equals VV_s, to avoid livelock, Ekko will choose a shard version from s and r following deterministic rules (e.g. choosing the shard version which exhibits a larger numerical value).

We implement book-keeping techniques [51] which maintain the shard version lists associated with different replicas. By applying both shard versions and book-keeping, Ekko can effectively reduce synchronisation-oriented network traffic. For example, in one of our production DLRSs, Ekko filters out 98% of shards in synchronisation.

Synchronisation with shard versions. We discuss how shard versions facilitate synchronisation. Ekko maintains the invariant $sv_1 \succeq sv_2$ only if shard knowledge VV_1 dominates VV_2 for the same shard s. Thus replica r needs to synchronise a shard with replica s only if $sv_r \not\succeq sv_s$. Furthermore, consider different replicas which have comparable shard versions for the same shard. Ekko prefers to synchronise with the one with the largest shard version because larger shard versions indicate a more refreshed version of parameters.

4.5 Implementation details

WAN optimisation. Ekko targets geo-distributed deployments, which comprise multiple intra-DC networks and an

inter-DC WAN. To improve its performance with such deployments, Ekko uses a *WAN-optimised model update dissemination strategy*. This strategy constructs a flexible communication topology for P2P synchronisation. It lets each DC elect a local leader for each shard using Zookeeper [31]. The leaders pull model updates from other DCs while other replicas pull updates from this leader. By doing so, Ekko allows a large proportion of synchronisation traffic to go through bandwidth affluent intra-DC networks and only a small of synchronisation traffic to go over WANs. Note that the implementation of the parameter synchronisation does not require a specific communication topology. Ekko can use other overlay topologies to improve synchronisation performance.

Failure tolerance. Ekko uses the request routers to tolerate failures. The routers decide the routes of client requests, and they detect the healthiness of replicas using heartbeats. Suppose a router speculates a replica failure (either fail-stop or fail-slow [30]). In that case, it prevents clients (inference servers and training servers) from requesting that replica. It also tracks the shard knowledge of replicas in the cluster. If a previously suspected failed replica recovers and sends heartbeats to the router, the router will instruct that replica to catch up with a sufficiently updated replica in the cluster. When the catching-up finishes, the router directs client requests to that replica. If a replica loses its state, it re-joins the cluster with a new id. Training servers stop sending parameter updates if they cannot contact the router for a given period, which achieves best-effort protection of model parameters from divergence in the case of having network partitions [5].

5 SLO Protection Mechanisms

Ekko allows model updates to reach parameter servers in inference clusters directly. This, however, raises two challenges for the SLOs of recommendation services: (i) network congestion can cause critical model updates to be delayed, and (ii) model updates based on a small batch of biased data can have detrimental impacts on inference results.

This section introduces mechanisms that protect inference SLOs from network congestion and biased updates. We first define the SLOs (see §5.1), describe an SLO-aware model update scheduler (see §5.2), and discuss an inference model state manager that handles biased updates (see §5.3).

5.1 SLOs in a DLRS

A DLRS has two major types of SLOs:

- **Freshness SLOs** measure the latency of including new content and users in model inference. They are vital for recommendation services, especially those interacting with users in real-time, e.g. TikTok and YouTube. For example, such services often need to capture the interests of new users in a timely manner so that they are sufficiently engaged; otherwise, they leave the recommendation applications due to the loss of interest. Improving the freshness SLOs usually leads to a better user experience. Also, new content will have better exposure, securing the prosperity of DLRSs.
- **Quality SLOs** measure user experience and engagement. They have immediate impacts on the profitability of a DLRS. Examples of such objectives include the number of viewed videos and user watching time.

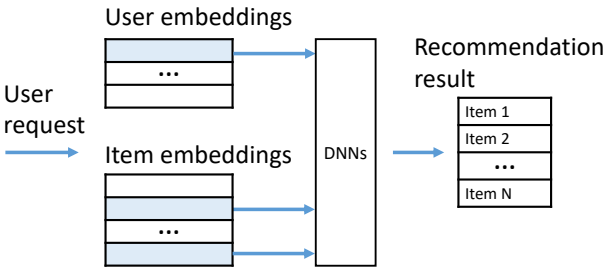


Figure 4: Overview of the inference process in Ekko.

Figure 4 describes how an inference server affects the freshness and quality SLOs. Once receiving a request, the inference server selects related user and item embeddings. It then aggregates the embeddings and sends an aggregated embedding to a DNN that returns the scores for recommendation items. The DLRS finally returns a list of items sorted by the scores. In this case, the freshness SLO is measured based on the latest timestamp of the recommended items (Ideally, this timestamp should be as close to the current time as possible). The quality SLO can be measured based on the viewing time of the items and how many items are clicked. In practice, Ekko maintains a large number of freshness and quality SLOs online. The implementations of such SLOs are contributed by DLRS application developers.

5.2 SLO-aware model update scheduler

Ekko prevents both freshness and quality SLOs from being affected by network congestion. This is achieved by an SLO-aware model update scheduler and an integration of this scheduler into P2P model update dissemination.

5.2.1 SLO-aware priorities for model updates

Ekko computes a set of priorities in scheduling model updates:

Update freshness priority. Ekko computes an update freshness priority p_u. This priority is designed based on the following observation. If a parameter has been created recently, it has a high priority; otherwise, it has a relatively lower priority. The reason for this is that newly created parameters have more significant impacts on inference results than those served for

an extended period. Suppose a user's embedding is unavailable in the inference server, but her request has arrived. In this case, the DLRS cannot answer this request, compromising quality SLOs. Another case is that if the embedding table does not include an item on the inference servers, the DLRS will not recommend this item, compromising freshness SLOs.

Update significance priority. Ekko computes an update significance priority p_g for each model update based on its gradient g. This priority is initially inspired by studies which showed how the gradient magnitude $|g|$ affects the inference results of a DNN [6, 28]. However, naively adopting the gradient magnitude is insufficient in Ekko. As a shared DLRS, Ekko multiplexes the updates from different models on a shared network. As a result, Ekko must have ways to compare gradient magnitudes that have different distributions. Therefore, we define $p_g = |g|/\overline{|g|}$, where $|g|$ denotes the 1-norm of g and $\overline{|g|}$ denotes the average gradient magnitude of recent model updates. Intuitively, this definition normalises gradient magnitudes, thus making them comparable.

Model priority. In a DLRS, models often receive inference requests at different rates, indicating their varied importance in measuring the overall satisfaction of SLOs. To consider this, Ekko allows the models that handle the majority of requests to be assigned with higher priorities compared to those that rarely receive requests. To this end, we define the model priority as $p_m = c_m / \sum_{i=1}^{M} c_i$, where c_m is the request count of model m and $\sum_{i=1}^{M} c_i$ denotes the total request count of all M models.

Combining priorities. We combine all the above priorities to compute the overall priority p of a model update as below:

$$p = (p_g + p_u)\, p_m$$

where the significance priority p_g and the freshness priority p_u have both been normalised so that they can be summed up. The sum is multiplied by the model priority p_m.

Note that Ekko does not require its users only to use the above priorities. Some Ekko users have custom priority definitions, including update count, update interval and the positions of parameters in embedding tables. These custom priorities are specific to certain DLRS workloads [69], and they are not generic enough to be included in a default setting. Ekko accommodates these custom priorities by supporting User-Defined-Functions (UDFs) in defining priorities.

5.2.2 Scheduler implementation

The model update scheduler computes the priority for each update once it is produced. It needs to ensure the cost of priority computation is negligible; otherwise, it can become a bottleneck in model updates. To achieve this, the scheduler offloads the maintenance of priority-related statistics (e.g. $\overline{|g|}$ and p_m for each model m) to a background thread. Moreover, to bound memory cost, it uses a quantile sketch (e.g. DDSketch [52]) that computes the k percentile priority p_k in a time window, where k is a ratio set by algorithm managers. Ekko executes user-defined priority computation using WebAssembly [27] to achieve efficient isolation among UDFs.

Algorithm 2: Priority-based synchronisation

```
1  Function UpdateSVV(SVV_other):
2      SVV.Merge(SVV_other);
3      TSVV.Merge(SVV);
4  Function WriteStoreParameter(p):
5      WriteIfVersionLarger(store, p);
6      EraseIfVersionNotSmaller(store_significant, p);
7  Function OnRecvPrioritisedSync(TSVV_other):
8      reply.TSVV ← TSVV;
9      for p ∈ (store ⋃ store_significant) do
10         if not TSVV_other.Dominate(p.sigv) then
11             reply.parameters.Add(p);
12         end
13     end
14     return reply;
15 Function PrioritisedSync():
16     reply ← OnRecvPrioritisedSync_other(TSVV);
17     for p ∈ reply.parameters do
18         if VersionLarger(store ⋃ store_significant, p)
             then
19             store_significant[p.name] ← p;
20         end
21     end
22     TSVV.Merge(reply.TSVV)
```

Integrating schedulers into parameter servers. To achieve the promise of priority scheduling, we must have ways of integrating the schedulers into the parameter servers which have enabled log-less P2P synchronisation. To this end, we propose the *significant version*, denoted as *sigv*, for each parameter and the *significant knowledge SVV* for each shard. Moreover, Ekko assigns each shard with a *transient significant parameter store* $store_{significant}$ and a corresponding *transient significant knowledge TSVV* to enable P2P synchronisation with priority scheduling.

Algorithm 2 describes the log-less P2P synchronisation augmented with priority schedulers. Suppose we have a model update from a replica. In this case, Ekko calculates p. If $p \geq p_k$, Ekko sets $sigv = v$, where v is the parameter version of this update; otherwise, *sigv* remains unchanged. Then, Ekko uses *sigv* to construct SVV_{other} and call the UPDATESVV function (line 1). In the case that Ekko does not apply priorities in synchronisation, replicas exchange *SVV* and execute the UPDATESVV function. On writing parameters into the persistent parameter store, Ekko prunes superseded parameters by executing the WRITESTOREPARAMETER function (line 4). Note that replicas estimate how long the model updates to reach themselves. Hence, when network congestion occurs, servers will have update time-outs. In this case, Ekko

uses the PRIORITISEDSYNC function (line 15) that triggers priority schedulers in synchronisation. Once receiving requests, replicas prefer to return parameters in significant parameter stores.

5.3 Inference model state manager

Ekko uses an inference model state manager to protect SLOs from detrimental model updates. This manager monitors inference models' healthiness (i.e. quality SLOs) and conducts low-latency model state rollback on demand.

5.3.1 Monitoring model healthiness

Ekko monitors model healthiness based on the following idea: for a DLRS application, it creates *baseline models* for its inference models. Baseline models process a small amount of user traffic (usually $< 1\%$). They are different from the online inference models because they carry delayed states. In other words, they are trained with previous training samples, usually several minutes earlier than the samples training the current inference model.

Ekko measures model healthiness based on metrics collected from inference servers and clients (e.g. user devices). To compute these metrics, Ekko defines a custom watermark and trigger [3]. Its state manager emits anomaly detection events only if confident (i.e. observing monitoring data for an extended period). Note that Ekko is not constrained to use specific anomaly detection algorithms. It supports custom anomaly detection algorithms, such as those often used with time-series data [61].

We model the transition of model states (i.e. healthy or not) as a replicated state machine [63], implemented within the model state manager. This manager evaluates and records model healthiness at a timestamp t by inspecting the healthiness-related metrics and the model update latency. The timestamp t monotonically increases. The manager makes judgements if the model state is *healthy*, *corrupted* or *uncertain*. When the manager is confident that changes have occurred in the model state (i.e. healthy or corrupted), it records this information in its replicated state. If the model state has corrupted, the manager re-directs client requests to alternative inference models (still healthy) and then launches a model state rollback.

5.3.2 Low-latency model state rollback

Ekko uses *witness servers* to roll back corrupted model states at low latency. The witness servers join replica synchronisation but they do not participate in model training. Unlike parameter servers, the witness servers (i) do not immediately flush updated parameters into parameter stores and (ii) do not run priority scheduling in synchronisation. More specifically, Ekko inserts the parameter updates that are not flushed yet into the logs. The logs are attached with the physical timestamp of synchronisation (denoted as t). If there are multiple synchronisation operations in a small time window, Ekko merges their logs to save space.

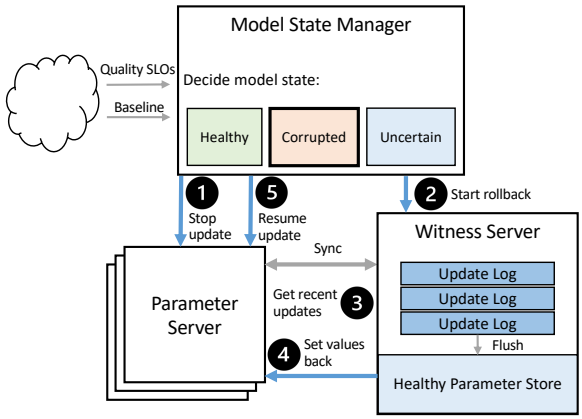


Figure 5: Inference model state manager.

The model state manager controls witness servers to launch state rollbacks. Suppose a model state is regarded as $t_{healthy}$ at the time t. In that case, witness servers find a timestamp t_{max} that meets two conditions: (i) it is $\leq t_{healthy}$ and (ii) it is not within any time interval where corrupted states have occurred. The witness servers then flush the logs which have the timestamps $\leq t_{max}$. The model state manager records this t_{max}, and t_{max} will be later used in witness servers for recovering a healthy model state. Following this way, we can ensure the parameter store $store_{healthy}$ always keep healthy model states on witness servers.

Rollback process. Figure 5 illustrates the process of rolling back a model state. Suppose a model is found to be corrupted. The model state manager first informs parameter servers to stop accepting training requests of this model (❶). It then instructs parameter servers to stop priority-based synchronisation, clears their $store_{significant}$, and resets $TSVV = SVV$. The manager then waits for the model shards on parameter servers and witness servers to converge. Later, the manager selects witness servers to initiate the state rollback (❷). We need to ensure recovered model shards can be used together. Hence, the manager selects shards from the $store_{healthy}$ on witness servers only if t_{max} of these shards are in a small time window.

A key design is that the witness servers will compare $store_{healthy}$ and its current state to find a state difference (❸). This difference is often small because of the locality in updated parameters. We thus only write the difference into the parameter servers to recover a state. We need to ensure the write operations can succeed. Hence, the written parameters are assigned with parameter versions that are larger than those currently on parameter servers (❹). After that, the man-

ager waits for the model shards to converge on parameter servers and witness servers. Finally, Ekko will recover a small amount of traffic on the recovered model. When this model's healthiness metrics go back to normal, the manager informs parameter servers to resume accepting requests (❺).

Note that if a witness server fails, its non-flushed update logs are discarded. This helps Ekko prevent potentially corrupted updates from being flushed. If a parameter server or a witness server fails (or re-joins the cluster), the rollback process will be re-executed.

6 Evaluation

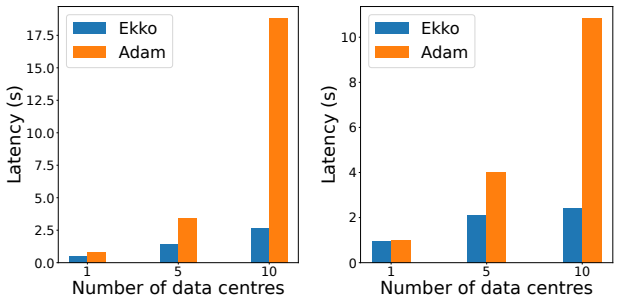
In this section, we evaluate the following aspects of Ekko through test-bed and in-production experiments: (i) The update latency of Ekko and its scalability with the number of data centres (§6.1.1); (ii) The update latency of Ekko in a heterogeneous-WAN (§6.1.1); (iii) The performance breakdown of optimisations implemented in Ekko (§6.1.2); (iv) The real-world latency and availability of Ekko in a large-scale production DLRS (§6.2.1); (v) The benefits of low-latency model updates in online services (§6.2.1); (vi) The effectiveness of using model update schedulers with a busy network (§6.2.2); and (vii) The latency of rolling back a model upon model corruption (§6.2.2).

Unless otherwise specified, the update latency is the maximum time difference between the time an update commits and the time this update becomes visible [68] in all replicas (failure-free scenarios). In all experiments, we measure the update latency and report its average across all updates.

6.1 Test-bed experiments

We conduct test-bed experiments in a 30-server cluster. Each server has a 24-core CPU, 64 GB RAM and a 5 Gbps network link. We group every three servers as a DC to emulate a multi-DC scenario, forming up to 10 DCs. We choose one of the DCs as the training-oriented DC, which receives model updates from a server (which acts as a DLRS client). We let other DCs be inference-oriented and connect them with the training-oriented DC. The inter-DC bandwidth is 4,800 Mbps (unless otherwise specified), emulating a WAN.

Our test-bed experiments comprise two workloads. The first workload trains a large ranking model typically used in our production environments. In this workload, we choose the shard size as 0.4 MB. The second workload trains the Wide & Deep model [10] using the Criteo Terabyte Click Logs [17] sorted chronologically. We initialise embedding tables using 21-day data logs. To ensure experiments are reproducible, we record model update traces and replay them during the experiments.



(a) Production workload (b) Criteo workload

Figure 6: Average model update latency.

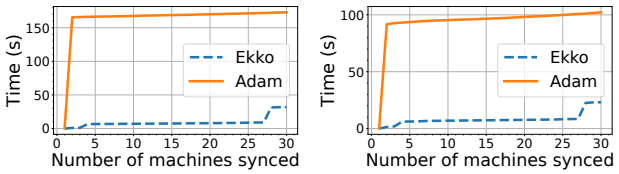
6.1.1 Update latency

We evaluate Ekko's update latency in a homogeneous WAN and a heterogeneous WAN. Both of these WANs are common in the real world. The first baseline is Adam [11] which is often used in parameter servers to synchronise model updates using the two-phase commit protocol. Our Adam implementation removes the waiting time between update broadcasts, thus improving network utilisation. The second baseline is Checkpoint-Broadcast which is the de-facto approach that applies model updates in DLRSs [1, 21]. We omit the experiments with general key-value stores, e.g. PaxosStore [73] and TiKV [29], which provide linearisability in writing operations. Our early adoption results show that these key-value stores achieve low writing throughput, orders of magnitude lower than what a production DLRS requires.

To make a fair comparison, Ekko and baselines all use DRAM for storage [57] and adopt the same primary-assignment and load-balancing schemes. We further ensure their dissemination are all network-bound and use the same numbers of shards.

Homogeneous WAN results. We first compare Ekko against Adam in the homogeneous WAN. We measure their latency with 1 DC (3 replicas), 5 DCs (15 replicas), and 10 DCs (30 replicas), respectively. Figures 6a and 6b show the results. As we can see, Ekko achieves significantly lower latency than Adam in both the production and Criteo workloads. More specifically, with the 10 DCs that run the production workload, Ekko achieves a 2.6-second latency, 7× lower than the 18.8-second latency achieved by Adam. We also observe that the performance gap between Ekko and Adam increases with more DCs. The reason is that Ekko has a scalable P2P synchronisation architecture. It also optimises its dissemination topology for a WAN. In contrast, Adam relies on the primary replica to send updates, constraining itself with the limited bandwidth available in the training DC.

We also compare Ekko against Checkpoint-Broadcast. According to our experimental results, Checkpoint-Broadcast takes more than 7 seconds to synchronise 4 GB of parameters in the WAN. The total parameters are 113 GB. With



(a) Production workload (b) Criteo workload

Figure 7: Update latency in a heterogeneous WAN.

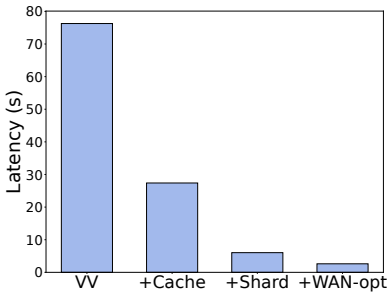


Figure 8: Performance breakdown.

10 DCs, the training DC needs to send $113 \times 9 = 1{,}017$ GB parameters to all other inference DCs. The training DC thus has to spend more than 29 minutes finishing the parameter broadcast (since the WAN has a 4,800 Mbps network link). This broadcast latency is orders of magnitude longer than the second-level latency (e.g. 2.6 seconds) achieved by Ekko.

Heterogeneous WAN results. We then evaluate Ekko and baselines in the heterogeneous WAN. In this WAN, we set inter-DC bandwidth to 256 Mbps by default. To introduce heterogeneity, we choose one link in between the training DC and another inference DC, and we set this link to 128 Mbps. The experiments run with 3 replicas per DC, for a total of 10 DCs. As shown in Figures 7a and 7b, Ekko is effective in mitigating slow heterogeneous links in both production and Criteo workloads. It allows replicas to synchronise at independent rates, preserving second-level synchronisation latency. Such low-latency performance shows the effectiveness of Ekko's log-less P2P synchronisation in alleviating the adverse effects of having heterogeneous network paths. On the contrary, Adam suffers from the slow paths in the WAN. As a result, it spends more than 150 seconds synchronising replicas in the production workload and 100 seconds in the Criteo workload.

Apart from Adam, we also considered other log-based synchronisation approaches, e.g. Multi-Paxos [9]. We could let these approaches aggregate updates (which arrive in a time interval) into a log entry to save bandwidth in using a WAN. These approaches, however, still suffer from the existence of heterogeneous links. This is because they choose the aggregation interval based on the slowest links in the network, under-utilising many other links.

6.1.2 Performance breakdown

We want to know the effectiveness of individual components in Ekko's synchronisation. We thus conduct a performance breakdown analysis for the production workload with 10 DCs. We first configure Ekko to only use shard knowledge (see §4.3) in synchronisation. This configuration is the baseline in this experiment, and it is equivalent to the Version Vector (VV) [50, 51] which is the state-of-the-art of P2P synchronisation.

Figure 8 shows the results. With only VV, Ekko needs 76.3 seconds to synchronise all parameters. After enabling update caches (§4.4.1), Ekko reduces the latency to 27.4 seconds (i.e. 2.8× speed-up). Diving into the update caches traces, we find out the caches achieve a 100% hit ratio in our production workload. Note that the total memories of a replica on our testbed servers are smaller (i.e. 10×) than those on our production servers, which means there are fewer parameters in a shard than in practical scenarios. With more parameters in a shard, VV will spend more time on synchronisation, while update caches can keep latency low.

Figure 8 also shows the effects of shard versions (§4.4.2). By further enabling shard versions, Ekko reduces the latency from 27.4 seconds to 6.0 seconds (i.e. 4.6× speed-up). This shows the effects of skipping non-updated shards to reduce network consumption incurred by synchronisation.

Finally, after enabling WAN optimisations (§4.5), Ekko further reduces the latency from 6.0 seconds to 2.6 seconds (i.e. 2.3× speed-up). This shows that P2P synchronisation must account for the bandwidth available on each link in a WAN. Otherwise, P2P synchronisation cannot deliver its full promise. In summary, enabling all components in Ekko leads to a total of 29.3× (i.e. 2.6 seconds vs. 76.3 seconds) speed-up in P2P synchronisation.

6.2 Production cluster experiments

We have deployed Ekko into production for over one year. The production environment comprises 4,600 servers spread across 6 geo-distributed DCs. By 2022, we have used Ekko to support a wide range of recommendation services, including short video recommendations, searching and advertisement. More than one billion users are using these services daily. In this section, we report Ekko's performance in this production environment.

6.2.1 Model updates

We collect traces from the production environment to analyse Ekko's performance in updating models. The production environment has hundreds of DLRS models (40 TB parameters or 250 billion key-value pairs in total). Each parameter shard ranges from 0.1 MB to 20 MB depending on model size. Ekko can execute 1 billion updates per second (i.e. 212 GB/s).

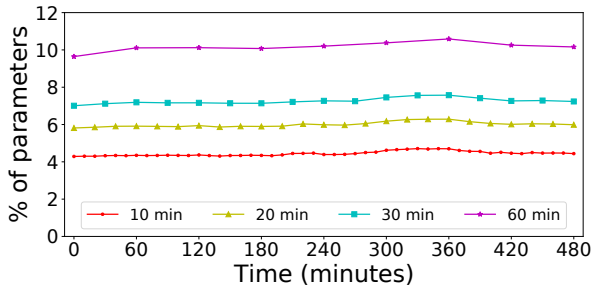


Figure 9: The proportions of updated parameters over time in different time intervals.

Regarding latency performance, Ekko spends 2.4 seconds synchronising the parameters in all DCs and 0.7 seconds in the training DC only. The synchronisation traffic accounts for only 3.0% of the total network traffic, reflecting the effectiveness of Ekko used as a background synchronisation service on parameter servers. Ekko's low-latency, high-throughput performance does not compromise system availability. Since its deployment, Ekko has achieved >99.999% availability for parameter reading and writing operations.

Update cache analysis. We are particularly interested in the performance of the update caches with various real-world recommendation services. Our traces show that: the update cache only needs to keep 0.13%-0.2% parameters in caches, and they can already achieve >99.4% hit ratios. These performance results verify that the update locality widely exists. In fact, our production recommendation services update 3.08% of the parameters per hour on average.

We choose an update-intensive DLRS model to demystify the update locality in the worse case. Figure 9 shows the proportions of updated parameters in a 480-minute window. This time window covers the busiest time of our production DLRSs in a day. We report the proportions with different time intervals. In a 10-minute interval, only 4.3% of parameters are updated, and this proportion is stable in the 480-minute time window. In a 60-minute interval, we observe a similar pattern, and the proportion only slightly increases to around 10%. In practice, many other models have fewer update workloads, and their proportions of updated parameters are lower than this model.

Benefits of low-latency model updates. We want to know if the low-latency model updates can actually improve the quality of recommendation services. To this end, we conduct a 15-day online A/B test [64] in a short video recommender service [65]. This service comprises a multi-stage pipeline [10, 15]. We conduct the experiment only in the ranking stage. We fork the ranking model: one as the experimental group and the other as the control group. Each group receives 1% of the total traffic for training and inference. We delay the data (i.e. event logs) used to train the model in the control group by 20 minutes through caching real-time logs in a distributed file system.

Our A/B-test results show that: compared to the control group, the experimental group exhibits a 3.82% increase in the proportion of fresh videos (posted within one hour) among all recommended videos. This means that the system recommends more fresh videos to users in the experimental group.

Moreover, the experimental group exhibits a 1.30% decrease in the proportion of users swiping through the video list as well as a 1.68% increase in the total time of browsing videos. These mean that users in the experimental group spend more time watching videos and are more interested in the recommended videos.

Finally, the experimental group exhibits a 2.17% increase in the percentage of users who clicked on comments. This means that user interaction in the experimental group increases. It is worth noting that the improvements in the range of 1%-3% are regarded as significant in a real-world multi-stage DLRS [10, 21, 71]. In fact, since enabling low-latency model updates in more stages in DLRSs, we have observed more significant improvements in recommendation quality.

6.2.2 SLO protection mechanisms

We also run A/B tests to evaluate the effectiveness of Ekko's SLO protection mechanisms.

SLO-aware model update scheduler. We fork the ranking model into an experimental group (where priority schedulers are enabled) and a control group. Each group has 1% of the training and inference traffic, and they are deployed into dedicated servers to avoid traffic interference. We monitor metrics that reflect freshness SLOs: the count of fresh videos (i.e. posted in the last one hour) in recommendation results. To emulate network congestion, we reduce the bandwidth available for model updates by 92%. The model update scheduler (i) uses the default priority computation rule (defined in §5.2.1) and (ii) sets the percentile priority k to 99 (k is defined in §5.2.2).

The A/B-test results show that, in the experimental group, Ekko reduces synchronisation traffic by 92% and keeps the latency of updating significant updates low. In contrast, the control group cannot distinguish model updates when sending them over a busy network. As a result, the control group delays SLO-critical updates, and it suffers from a 2.32% drop in its SLO metric. Such a drop is significant in practice because this SLO metric is a key factor that decides the profit of a DLRS.

Online model state rollback. We evaluate the latency of rolling back a model state online. We compare Ekko with the checkpoint-recovery approach. To make a fair comparison, we let the rollback latency exclude (i) the time of collecting SLO metrics in Ekko and (ii) the time of waiting for diverged parameters to converge. We deploy 5 witness servers. For each witness server, we allocate 113 GB parameters and 800 Mbps network bandwidth.

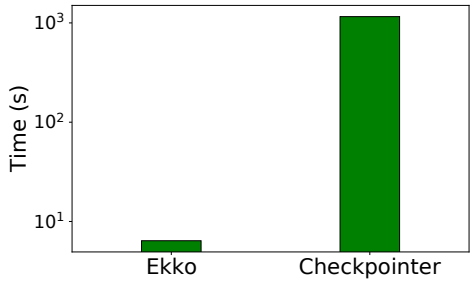


Figure 10: Model state rollback time.

During the experiment, we notify Ekko's model state manager to roll back the state of a DLRS model to a version that is 1 minute earlier. The manager then notifies all witness servers to identify the parameters updated in the last 1 minute. The witness servers thus only reload the difference between the current state and the earlier state. Hence, the entire rollback operation takes only 6.4 seconds to complete. In contrast, the checkpoint-recovery approach is agnostic to the recent updates to the model state. As a result, it has to reload the entire state, taking 1,157 seconds to complete (180× slower than Ekko).

7 Related Work

Data replication systems. The parameter synchronisation problem explored in Ekko is related to prior work on data replication. Existing data replication systems often explore how to leverage the characteristics of applications to improve their latency performance in replicating data [13, 40, 45, 53]. For example, Egalitarian Paxos [53] exploits the low interference rate of state machine commands, Gemini [40] leverages mixed consistency operations, and COPS [45] and PNUTS [13] exploit the tolerance of relaxed consistency in Internet services. Unlike these systems, Ekko leverages the DLRS-specific model update locality and the eventual consistency model to speed up the synchronisation of model parameters (instead of generic data), making Ekko unique in the design space.

Bandwidth saving techniques in ML systems. The problem of prioritising model updates relates to bandwidth saving techniques in distributed ML systems. Such techniques often involve gradient compression [4, 6, 28, 44] which prioritises large gradients in a busy network, with an anticipation that these large gradients have significant impacts on the final accuracy of a trained model. Unlike these techniques, Ekko targets model inference scenarios where people care about numerous inference SLO metrics instead of the model's accuracy only. Hence, Ekko does not rely on gradient magnitude solely. It further considers model freshness and priority in scheduling model updates.

SLO-aware scheduling in ML systems. Being aware of SLOs in scheduling has been explored in prior ML systems. Model serving systems often treat inference latency as the primary SLO to guide the scheduling of inference-related computation tasks [16, 25, 70]. Model training systems, e.g. Pollux [59] and KungFu [48], use ML-specific SLOs, e.g. training goodput and gradient statistics, to decide how to schedule training workers. Compared to these systems, Ekko sheds light on freshness and quality SLOs. It enables the use of these SLOs in scheduling model updates.

8 Conclusion

This paper proposes Ekko, a novel DLRS that enables massive model parameters to be updated at the second-level latency. Ekko has an efficient P2P model update algorithm which can coordinate billions of model updates to be efficiently disseminated to replicas in geo-distributed data centres. It further has SLO protection mechanisms that protect model states from being affected by network congestion and detrimental model updates online. Experimental results show that Ekko is orders of magnitudes faster than state-of-the-art DLRSs, indicating the effectiveness of its novel designs.

Acknowledgements

We sincerely thank our shepherd Miguel Castro and the OSDI reviewers for their insightful suggestions. This paper presents a multi-team effort previously known as WeChat Parameter Server (WePS). Part of this work is supported by gift funding from Tencent.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Dissecting the performance of strongly-consistent replication protocols. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1696–1710, 2019.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven

Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[4] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cedric Renggli. The convergence of sparsified gradient methods. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[5] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of Network-Partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, Carlsbad, CA, October 2018. USENIX Association.

[6] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. *Gradient Compression Supercharged High-Performance Data Parallel DNN Training*, page 359–375. Association for Computing Machinery, New York, NY, USA, 2021.

[7] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 374–388, 2019.

[8] ByteDance. TikTok. https://www.tiktok.com/, 2021. Accessed on 2021-12-08.

[9] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 398–407, New York, NY, USA, 2007. Association for Computing Machinery.

[10] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[11] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[12] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John K. Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In Andrew Birrell and Emin Gün Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 37–48. USENIX Association, 2013.

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

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

[15] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, page 191–198, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[17] CRITEO. CRITEO Terabyte Click Logs. https://labs.criteo.com/2013/12/download-terabyte-click-logs/, 2022. Accessed on 2022-05-04.

[18] Gabriel de Souza Pereira Moreira, Sara Rabhi, Jeong Min Lee, Ronay Ak, and Even Oldridge. *Transformers4Rec: Bridging the Gap between NLP and Sequential / Session-Based Recommendation*, page 143–153. Association for Computing Machinery, New York, NY, USA, 2021.

[19] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[20] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

[21] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, Renton, WA, April 2022. USENIX Association.

[22] Facebook. Instagram. https://www.instagram.com/, 2021. Accessed on 2021-12-11.

[23] Weihao Gao, Xiangjun Fan, Chong Wang, Jiankai Sun, Kai Jia, Wenzhi Xiao, Ruofan Ding, Xingyan Bin, Hui Yang, and Xiaobing Liu. Learning an end-to-end structure for retrieval in large-scale recommendations. In Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong, editors, *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, pages 524–533. ACM, 2021.

[24] Google. Youtube. https://www.youtube.com/, 2021. Accessed on 2021-12-06.

[25] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[26] Huifeng Guo, Ruiming TANG, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: A factorization-machine based neural network for ctr prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1725–1731, 2017.

[27] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017.

[28] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 629–647, Boston, MA, March 2017. USENIX Association.

[29] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: A raft-based htap database. *Proc. VLDB Endow.*, 13(12):3072–3084, aug 2020.

[30] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal, editors, *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, pages 150–155. ACM, 2017.

[31] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[32] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, Liqin Zhao, Zhi Wang, Peng Sun, Yu Zhang, Di Zhang, Jinhui Li, Jian Xu, Xiaoqiang Zhu, and Kun Gai. Xdl: an industrial deep learning framework for high-dimensional sparse data. *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, 2019.

[33] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[34] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. *Proceedings of the VLDB Endowment*, 12(11).

[35] Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527*, 2016.

[36] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[37] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 19–35. USENIX Association, July 2021.

[38] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard manager: A generic shard management framework for geo-distributed applications. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 553–569. ACM, 2021.

[39] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *arXiv preprint arXiv:1903.12287*, 2019.

[40] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-Replicated systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, October 2012. USENIX Association.

[41] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association.

[42] Xiang Li, Kaixuan Huang, Wenhao Yang, Shusen Wang, and Zhihua Zhang. On the convergence of fedavg on non-iid data. In *International Conference on Learning Representations*, 2020.

[43] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186. USENIX Association, November 2020.

[44] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[45] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery.

[46] Luo Mai, Chuntao Hong, and Paolo Costa. Optimizing network performance in distributed machine learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[47] Luo Mai, Alexandros Koliousis, Guo Li, Andrei-Octavian Brabete, and Peter Pietzuch. Taming hyperparameters in deep learning systems. *ACM SIGOPS Operating Systems Review*, 53(1):52–58, 2019.

[48] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. KungFu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 937–954. USENIX Association, November 2020.

[49] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment*, 11(10):1303–1316, 2018.

[50] Dahlia Malkhi, Lev Novik, and Chris Purcell. P2p replica synchronization with vector sets. *SIGOPS Oper. Syst. Rev.*, 41(2):68–74, April 2007.

[51] Dahlia Malkhi and Doug Terry. Concise version vectors in winfs. In Pierre Fraigniaud, editor, *Distributed Computing*, pages 339–353, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[52] Charles Masson, Jee E. Rim, and Homin K. Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proc. VLDB Endow.*, 12(12):2195–2205, 2019.

[53] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery.

[54] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[55] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: continuously optimized region-wide datacenter resource allocation. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 505–520. ACM, 2021.

[56] Even Oldridge, Julio Perez, Ben Frederickson, Nicolas Koumchatzky, Minseok Lee, Zehuan Wang, Lei Wu, Fan Yu, Rick Zamora, Onur Yilmaz, et al. Merlin: A gpu accelerated recommendation framework. In *Proceedings of IRS*, 2020.

[57] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, jul 2011.

[58] D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.

[59] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[60] General Data Protection Regulation. Regulation eu 2016/679 of the european parliament and of the council of 27 april 2016. *Official Journal of the European Union*, 2016.

[61] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. Time-series anomaly detection service at microsoft. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 3009–3017, New York, NY, USA, 2019. Association for Computing Machinery.

[62] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, mar 2005.

[63] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[64] Diane Tang, Ashish Agarwal, Deirdre O'Brien, and Mike Meyer. Overlapping experiment infrastructure: more, better, faster experimentation. In Bharat Rao, Balaji Krishnapuram, Andrew Tomkins, and Qiang Yang, editors, *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, pages 17–26. ACM, 2010.

[65] Tencent. WeChat. https://www.wechat.com/, 2022. Accessed on 2022-05-06.

[66] Tensorflow. TensorFlow Recommenders Addons. https://github.com/tensorflow/recommenders-addons, 2021. Accessed on 2021-12-08.

[67] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.

[68] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1), jun 2016.

[69] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: Memory-efficient continual learning for large-scale real-time recommendations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[70] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 389–405. USENIX Association, April 2021.

[71] Zhiqiang Xu, Dong Li, Weijie Zhao, Xing Shen, Tianbo Huang, Xiaoyun Li, and Ping Li. *Agile and Accurate CTR Prediction Model Training for Massive-Scale Online Advertising Systems*, page 2404–2409. Association for Computing Machinery, New York, NY, USA, 2021.

[72] Yuchao Zhang, Junchen Jiang, Ke Xu, Xiaohui Nie, Martin J. Reed, Haiyang Wang, Guang Yao, Miao Zhang,

and Kai Chen. Bds: A centralized near-optimal overlay network for inter-datacenter data replication. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[73] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. Paxosstore: High-availability storage made practical in wechat. *Proc. VLDB Endow.*, 10(12):1730–1741, aug 2017.

[74] Lixin Zou, Long Xia, Zhuoye Ding, Jiaxing Song, Weidong Liu, and Dawei Yin. Reinforcement learning to optimize long-term user engagement in recommender systems. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2810–2818, New York, NY, USA, 2019. Association for Computing Machinery.

FAERY: An FPGA-accelerated Embedding-based Retrieval System

Chaoliang Zeng[1*] Layong Luo[2] Qingsong Ning[2] Yaodong Han[2] Yuhang Jiang[2] Ding Tang[1*]
Zilong Wang[1*] Kai Chen[1] Chuanxiong Guo[2]
[1]*Hong Kong University of Science and Technology* [2]*ByteDance*

Abstract

Embedding-based retrieval (EBR) is widely used in recommendation systems to retrieve thousands of relevant candidates from a large corpus with millions or more items. A good EBR system needs to achieve both high throughput and low latency, as high throughput usually means cost saving and low latency improves user experience. Unfortunately, the performance of existing CPU- and GPU-based EBR are far from optimal due to their inherent architectural limitations.

In this paper, we first study how an ideal yet practical EBR system works, and then design FAERY, an FPGA-accelerated EBR, which achieves the optimal performance of the practically ideal EBR system. FAERY is composed of three key components: It uses a high bandwidth HBM for memory bandwidth-intensive corpus scanning, a data parallelism approach for similarity calculation, and a pipeline-based approach for K-selection. To further reduce hardware resources, FAERY introduces a filter to early drop the non-Top-K items. Experiments show that the degraded FAERY with the same memory bandwidth of GPU still achieves 1.21×-12.27× lower latency and up to 4.29× higher throughput under a latency target of 10 ms than GPU-based EBR.

1 Introduction

Recommendation systems have gained significant adoption in many online services [11, 12, 18, 38]. To make a recommendation from a large corpus containing millions of candidate items, industrial large-scale recommendation systems are usually divided into two layers, namely retrieval and ranking, as shown in Figure 1. Retrieval quickly selects thousands of relevant items from the large corpus with simple algorithms, while ranking utilizes sophisticated algorithms to sort the retrieval results more precisely, and then chooses dozens out of the sorted items.

Real-world retrieval systems conduct multi-channel retrieval [26, 39, 43]: It leverages different strategies in separate channels to retrieve different candidates, which are then merged and filtered to generate the final retrieval result. Among the multi-channel retrieval strategies, embedding-based retrieval (EBR) gains increasing popularity [12, 18, 20, 25, 38, 42]. EBR represents user queries and candidate items with semantic embedding vectors (embedding for short) using representation learning [9], and converts the retrieval problem into a similarity search problem in the embedding space. In particular, an EBR algorithm, as shown in Listing 1, involves *scoring*, which scans the corpus to get all items and calculates a similarity score (e.g., via inner product) between every item embedding and the given query embedding, and *K-selection*, which returns the Top-K items based on their similarity scores. The returned Top-K items of EBR are usually sorted [25, 42], to simplify merging and filtering retrieval candidates from multiple channels.

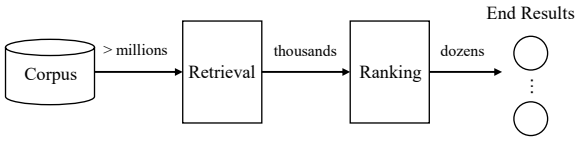


Figure 1: A typical recommendation system. Retrieval selects thousands of candidate items from a large corpus, and ranking further chooses dozens from the retrieval results.

The performance of such EBR systems is important. On the one hand, increasing the throughput of every EBR server reduces the overall server cost, as fewer servers are required to serve a target number of queries per second (QPS). On the other hand, decreasing the latency of each EBR server reduces the retrieval time, which can either shorten user's overall waiting time or leave more time for ranking computation to get better recommendation results [11]. Therefore, *latency-bounded throughput* becomes a critical metric for EBR systems.

To achieve high latency-bounded throughput, we characterize the EBR algorithm shown in Listing 1 and derive a practically ideal EBR hardware architecture (§2.2). Specifically, corpus scanning (line 3) is a memory-intensive operator which requires both large external memory capacity and high memory bandwidth. Similarity calculation (line 4) and K-selection (line 6) are both compute-intensive. They should match the memory bandwidth with a data-parallel architecture across multiple operator instances. Moreover, to overlap communications with computations among steps or operators, both inside K-selection and the entire EBR data flow require pipeline parallelism. Then, we extend the ideal architecture to support batch queries, by sharing corpus scanning among queries in a batch and providing separate compute pipelines

[*] This work is done while Chaoliang Zeng, Ding Tang, and Zilong Wang are interns in ByteDance.

```
# Scoring
for i in corpus_size:
    item_emb = corpus[i] # corpus scanning
    scores[i] = sim_calc(user_emb, item_emb) # similarity calc
# K-selection
ret_items = topk(scores) # returns the sorted top_k items
```

Listing 1: Simplified EBR algorithm for a single user query.

to serve different queries in the batch in parallel. As a result, the ideal architecture achieves the optimal query latency, and scales the latency-bounded throughput linearly with the batch size.

By comparing existing CPU- and GPU-based EBR with the ideal architecture, we realize that, unfortunately, none of the existing approaches achieve the optimal performance due to their inherent architectural limitations (§2.3). First, despite large memory capacity, CPU does not perform well in corpus scanning due to low memory bandwidth, and fails to well support the desired parallelism paradigms simultaneously due to the limited number of cores. Second, although GPU provides higher memory bandwidth and massive compute cores for data parallelism, GPU is not optimized for pipeline parallelism required by K-selection and the entire EBR data flow due to explicit resource boundaries.

We observe that FPGA, a programmable hardware device readily available in some hyper-scale cloud providers [10, 14, 41, 45], has all the desired properties of the practically ideal EBR architecture. Some modern FPGAs are equipped with large high bandwidth memory (HBM), ideal for corpus scanning. Moreover, FPGAs provide sufficient on-chip memories and fully programmable compute elements to enable appropriate parallelism paradigms for various operators (§2.4).

We exploit the above observations to design FAERY (§3), an **F**PGA-**A**ccelerated **E**mbedding-based **R**etrieval s**Y**stem, which is an embodiment of the ideal EBR architecture and achieves high performance. Specifically, FAERY stores the corpus in FPGA's HBM, which provides high bandwidth for the memory bandwidth-intensive corpus scanning. FAERY leverages a corpus manager to maximize the HBM bandwidth utilization in runtime while preserving memory-efficient storage and enabling online corpus update. FAERY follows the ideal architecture to design similarity calculation with data parallelism and K-selection with pipeline parallelism. Different from the ideal architecture, FAERY needs only a single K-selection pipeline, and adds a filter in front of it to significantly lower its throughput requirement, based on a unique property observed in the K-selection pipeline. The filter optimization lowers the resource requirements of FAERY compared with the ideal architecture by eliminating multiple K-selection pipelines.

The above ideas make a single FPGA-based EBR accelerator perform well. To further enhance its capabilities, multiple such accelerator cards can be inserted into a FAERY server (§4) and work together. When a corpus can fit into a single card, we can scale the aggregate query throughput by replicating the corpus among multiple cards. When the corpus is too large to fit into a single card, we can shard it evenly among multiple cards. FAERY supports both the *replication* and *sharding* modes and leverages a software front-end to dispatch queries and to merge retrieval results for multiple accelerator cards.

We have implemented a fully functional FAERY prototype with Xilinx FPGA cards (§5). Experiments (§6) show that the degraded FAERY with the same memory bandwidth of GPU achieves 1.21×-12.27× lower latency and up to 4.29× higher throughput under a latency target of 10 ms than an EBR system accelerated by Nvidia T4 GPU.

This paper makes the following contributions:

- We study the EBR algorithm from the first principles and derive a practically ideal EBR architecture to achieve the optimal query latency and to scale the latency-bounded throughput linearly with the batch size, constrained by hardware resources. We further identify the performance bottlenecks of CPU- and GPU-based EBR using the ideal EBR architecture as a reference (§2).
- We design FAERY, a domain specific accelerator (DSA) for EBR. FAERY arranges its key components: corpus scanning, similarity calculation, and K-selection in a perfect pipeline, and accelerates these components using appropriate data and/or pipeline parallelisms. FAERY is an embodiment of the ideal EBR architecture, with balanced filtering and buffering which matches the capability of parallel similarity score calculations with a single K-selection pipeline, based on a thorough analysis (§3 and §4).
- We implement FAERY using FPGA, evaluate its performance, and quantify its advantages over CPU- and GPU-based EBR systems, respectively (§5 and §6).

2 Background & Motivation

2.1 EBR Algorithms: KNN vs. ANN

EBR represents user queries and candidate items with embeddings, and converts the retrieval problem into a K-Nearest Neighbor (KNN) or an Approximate Nearest Neighbor (ANN) search problem in the vector space [20]. KNN-based EBR searches the **accurate** k-nearest item embeddings from the corpus, while ANN-based EBR retrieves the **approximate** k-nearest item embeddings, by sacrificing accuracy for efficiency using techniques such as indexing (e.g., IVF [34] and HNSW [28]) and quantization (e.g., PQ [21]). The tradeoff between accuracy and efficiency in various ANN algorithms is well studied in [8].

CPU provides limited memory bandwidth and computing power, so that it is challenging for CPU to perform KNN search on a large corpus due to the tremendous costs of memory accesses and computations. As a result, ANN search is

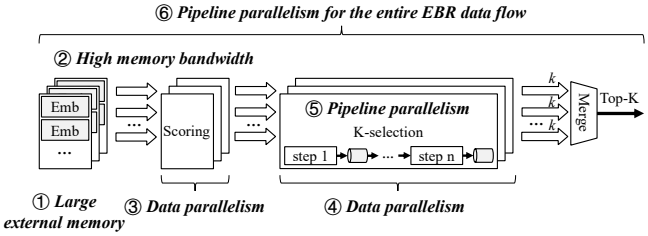


Figure 2: A practically ideal EBR architecture with the batch size of 1. It has the following properties: ① large external memory for corpus store, ② high memory bandwidth for corpus scanning, ③ data parallelism for similarity calculation, ④ data parallelism among multiple K-selection instances, ⑤ pipeline parallelism within a K-selection instance, and ⑥ pipeline parallelism for the entire EBR data flow.

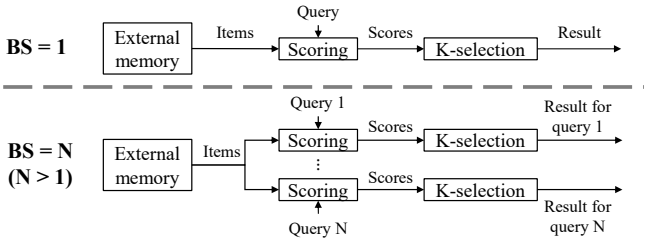


Figure 3: A practically ideal EBR architecture with the batch size of N, where the throughput scales linearly with the batch size N, while the latency remains the same as shown in Equation 1.

widely applied in CPU-based EBR in the industry. In contrast, accelerators, e.g., GPU and FPGA, provide much higher memory bandwidth and computing power, so that KNN search is usually adopted by these accelerators to trade memory bandwidth and computing power for higher accuracy and thus better recommendation quality.

To simplify discussion and comparison, we use the same KNN search (shown in Listing 1) for EBR on all platforms (CPU, GPU, and FPGA) in this paper, but our analysis results and acceleration ideas apply to ANN as well, as ANN shares similar characteristics and bottlenecks with KNN, just to different extents.

2.2 Practically Ideal EBR Architecture

To maximize latency-bounded throughput, an ideal architecture should first achieve minimal latency for each individual query (equivalent to maximal throughput with the batch size of 1), and then scale the throughput linearly with increasing batch sizes while preserving the consistent minimal latency.

In a theoretically ideal architecture, for each query, we do similarity calculation with ALL item embeddings in parallel and finish this operator in $O(1)$ time, followed by a perfect K-selection to match the parallelism. This is obviously impractical, as it requires millions of item accesses and millions of similarity calculation (e.g., inner product) operators in parallel, not to mention the design challenge of K-selection to match that extreme parallelism. A practically ideal EBR should take into account both realistic hardware constraints and the EBR characteristics which we discuss below.

Corpus scanning (line 3) is a memory-intensive operator. The size of an industrial corpus is up to several GBs [19], and scanning such a large corpus incurs millions of memory accesses for a single query. Thus, corpus store and scanning require large external memory and high memory bandwidth.

Similarity calculation (line 4) is a compute-intensive operator, which calculates similarity scores between the user query and all item embeddings. As the calculations for different item embeddings are independent, an ideal architecture should perform similarity calculation with data parallelism to match the throughput of corpus scanning.

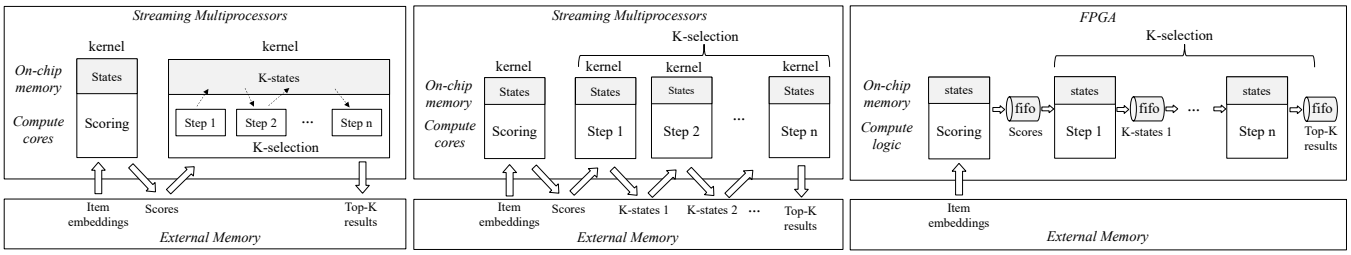
K-selection (line 6) is another compute-intensive operator. To match the throughput of multiple similarity calculation instances, K-selection requires data parallelism with multiple instances as well. Inside a single instance, K-selection can be realized by various algorithms [23, 33], among which a common practice is to partition this complex task into multiple steps, and organizes them in a pipelined manner.

Based on these characteristics, a practically ideal EBR architecture for optimal latency should have a large and high-bandwidth memory for corpus store and scanning, appropriate parallelisms for EBR operators to match their throughput to the memory bandwidth, and a perfect overlap among communications and computations of operators in the entire pipeline to minimize latency. Figure 2 describes a practically ideal EBR architecture with the batch size of 1 and its desired properties. The minimal query latency of this architecture is:

$$latency = \frac{S}{B} + C, \tag{1}$$

where S is the corpus size, B is the external memory bandwidth, and C is a constant delay, i.e., the pipeline latency, which is the time it takes for the last embedding going throughout the pipeline. Thus, the maximal throughput is $1/latency$ queries per second (QPS) with the batch size of 1.

The ideal architecture can be extended to support batch queries to increase latency-bounded throughput linearly, as shown in Figure 3. The key is to share corpus scanning among multiple queries in a batch (i.e., scan the corpus only once in each batch), and process multiple queries with separate compute pipelines in a data-parallel manner. In this way, the latency remains constant as shown in Equation 1, and the latency-bounded throughput scales linearly with the number of batched queries. In practice, the batch size cannot be increased unlimitedly due to resource constraints, and hence the maximum latency-bounded throughput will be bounded by the available hardware resource of the chosen platform.



(a) GPU-based EBR (GPU-o, K-selection with on-chip memory)

(b) GPU-based EBR (GPU-e, K-selection with external memory)

(c) FPGA-baesd EBR

Figure 4: Comparison between GPU- and FPGA-based EBR architectures. (a) GPU-o is a GPU-based EBR that stores intermediate states of K-selection in on-chip memory [23] and maintains corpus and scores in external memory. It suffers from heavy state maintenance cost and small k values; (b) GPU-e is a GPU-based EBR that moves all intermediate states of K-selection to external memory [33, 36], requiring multiple passes over the external memory; (c) is an FPGA-based EBR which stores only corpus in external memory, traverses external memory only once, and keeps the computation and communication of other operators fully on chip and in a streaming pipeline.

2.3 Existing EBR Architectures

Using the ideal EBR architecture as a reference, we analyze existing CPU- and GPU-based EBR architectures, and show that their performance are both sub-optimal due to inherent architectural limitations.

2.3.1 CPU-based EBR

Datacenter CPUs are equipped with large DDR memory (hundreds of GBs), able to store a very large corpus with millions or more item embeddings. However, CPU-based EBR does not perform well due to the following reasons.

Low memory bandwidth violates ② in Figure 2. The theoretical DDR memory bandwidth of a CPU is proportional to the limited number (typically 2∼8) of DDR channels [3], and the memory bandwidth utilization driven by a CPU is not high. Taking the server used in our evaluations (§6) as an example, a CPU with six DDR channels provides a theoretical maximum bandwidth of 140.8 GB/s, and an empirical upper bound of only 78 GB/s measured with Intel MLC [1]. The low memory bandwidth (B) significantly increases the first part (S/B) of Equation 1.

Limited number of CPU cores cannot support ③-⑥ and batch queries, simultaneously. A CPU contains dozens of processor cores that can be flexibly used for data parallelism, pipeline parallelism, and/or batch processing. However, due to the limited number of cores, CPU-based EBR fails to support all the above features well simultaneously, where the number of cores desired is the product of the number of memory channels, the number of pipeline stages, and the batch size as shown in Figure 2 and Figure 3. The poor support of data parallelism and pipeline parallelism results in throughput mismatch and imperfect overlapping among operators, leading to an increase on the second part (C) of Equation 1 as well as a sub-linear throughput increase with batch queries.

2.3.2 GPU-accelerated EBR

Compared with CPU, GPU provides high external memory bandwidth (e.g., Nvidia T4 [2] provides 300 GB/s bandwidth with GDDR6), and massive lightweight SIMT (Single Instruction Multiple Threads) cores optimized for data parallelism. Although GPU provides a smaller memory capacity (e.g., $16-80$ GB in a typical GPU and $128-640$ GB in a holistic server with 8 GPU cards), the size is still large enough to store the corpora in most recommendation services. For example, given a typical embedding size of 256 bytes, 128 GB memory can store more than 500M items that can meet the requirements of most recommendation systems [11,12,16,40]. These strengths inspire the design of GPU-accelerated EBR [23, 46] to achieve higher performance.

However, the performance of these GPU-based EBR systems are still sub-optimal, as GPU is not optimized for pipeline parallelism. GPU consists of a large number of *streaming multiprocessors* (SM), each of which contains exclusive on-chip memory and compute cores. Communication between SMs or kernels[1] is only possible via external memory, and the available on-chip memories for a single SM are very limited (e.g, 304 KB in Nvidia T4). These restrictions make GPU-based EBR not perfectly pipelined, leading to an increase on the second part (C) of Equation 1.

Inter-operator communication via external memory violates ⑥. Different EBR operators are organized as separate kernels. The similarity scores generated by scoring kernels are transmitted to the K-selection kernels via the external memory, as shown in Figure 4a and Figure 4b. The explicit kernel boundaries make it difficult to exploit pipeline parallelism

[1] A kernel is a function executed on GPU, which realizes a data-parallel portion of an application. An operator may consist of one or multiple kernels.

across EBR operators to overlap perfectly communication and computation [24, 47].

Existing K-selection pipelines violate ⑤. Existing GPU-based K-selection algorithms can be classified into the following two categories.

K-selection with on-chip memory (e.g., `WarpSelect` [23]), denoted as GPU-o, as shown in Figure 4a, fuses all K-selection sub-steps into a single kernel to avoid cross-kernel overhead, and keeps all K-states in on-chip memory. However, maintaining all K-states on chip and executing these steps in the SIMT cores introduce non-trivial computation overhead (e.g., per-thread queue sorting, sorted queues merging, and thread synchronization [23]), resulting in poor latency. To show this overhead, we measure Faiss [23], which adopts `WarpSelect`, with a 4M corpus and the same setting as §6. The result shows that the K-selection operator consumes up to 80.4% of the total time. Moreover, given the limited on-chip memory size of each SM, it fails to support a large k value, i.e., at most 2048 in this setting.

K-selection with external memory (e.g., `RadixSelect` [33, 36]), denoted as GPU-e, as shown in Figure 4b, implements different K-selection sub-steps as separate kernels, and transmits intermediate data among kernels via the external memory. As a result, the K-selection operator has to access the external memory multiple passes (well studied in [33]), leading to sub-optimal K-selection performance, which will become worse with a larger batch size due to heavy bandwidth contention on external memory. With the same setting mentioned above, the query latency of `RadixSelect` is increased by 12.36× when the batch size is increased from 1 to 16.

2.4 FPGA Opportunities

We observe that FPGA has the following properties that meet the requirements of the ideal EBR architecture.

- Similar to GPU, high-end FPGAs are equipped with HBM of large capacity (typically 8 to 32 GB). A typical HBM is a stack of 32 parallel DRAM channels (versus up to 8 DDR channels in a CPU), providing parallel memory accesses and thus high bandwidth (460 GB/s), which fundamentally eliminates the biggest memory bandwidth bottleneck in CPU-based EBR.
- Unlike GPU with exclusive and small on-chip memories for each SM, FPGA provides sufficient on-chip memories (dozens of MB in total), which are accessible to all compute elements. This could be leveraged to overcome the problems of GPU-based EBR as discussed in §2.3.2. Unlike GPU with SIMT cores optimized only for data parallelism, the massive compute elements and interconnects among them in FPGA are fully programmable, so that they can be orchestrated in any parallelism strategy (data parallelism or pipeline parallelism).

Desired features in ideal arch.	CPU	GPU	FPGA
large memory capacity	✔	✔	✔
high memory bandwidth		✔	✔
data parallelism	△	✔	✔
pipeline parallelism	△		✔
batch queries with low latency	△	△	✔

Table 1: EBR architecture comparison among CPU, GPU, and FPGA. ✔ means perfect support, while △ means limited support.

Table 1 summarizes the architecture comparison of CPU, GPU, and FPGA for EBR. Based on FPGA's advantages, we can design an FPGA-based EBR pipeline similar to that in Figure 4c: It traverses the HBM only once, passes intermediate data between operators via on-chip memory, and overlaps communications with computations of operators via careful pipeline designs. In this way, FPGA-based EBR has the potential to approach the optimal performance (Equation 1). The design details of such a system, named FAERY, are presented in the following sections.

3 FAERY Accelerator

We design the FAERY accelerator by following the most desired properties of the ideal EBR architecture, with some additional optimizations. Figure 5 presents the architecture of the FAERY accelerator, with a few major components including HBM, corpus manager, similarity calculation, filter, and K-selection. FAERY stores the corpus in HBM and uses the corpus manager (§3.1) for corpus scanning and update. FAERY applies data parallelism across multiple similarity calculation units (§3.2), and pipeline parallelism within K-selection (§3.3). Different from the ideal EBR architecture, FAERY does not need multiple K-selection pipelines with data parallelism, thanks to a new filter operator (§3.4) inserted before the K-selection pipeline to lower its throughput requirement. This optimization lowers the resource overhead compared with the ideal architecture. The above operators are perfectly pipelined and overlapped, and the resulting data streams are shown in §3.5.

3.1 Corpus Manager

FAERY stores the corpus in HBM and uses the corpus manager to perform corpus scanning and update. The corpus manager is designed to meet two objectives toward high bandwidth utilization of HBM: maximizing *single-channel performance* and maximizing *multi-channel parallelism*.

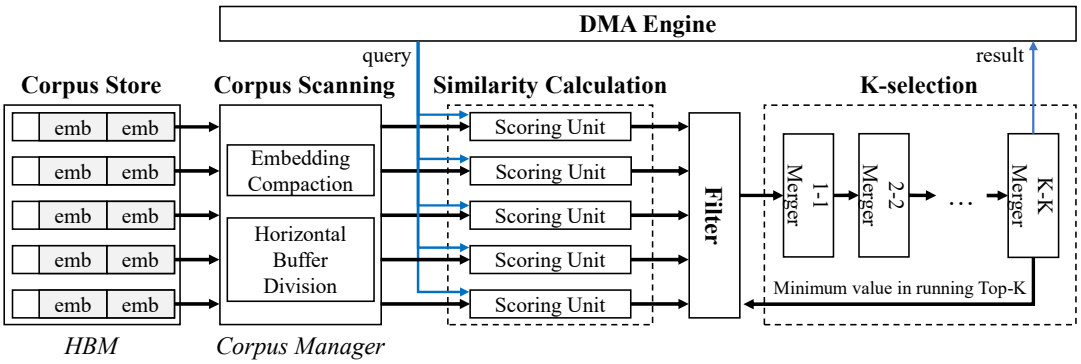


Figure 5: FAERY accelerator architecture with the batch size of 1. It stores item embeddings in high bandwidth memory (HBM), uses a corpus manager for corpus scanning and update, and applies appropriate parallelism paradigms for different key operators: data parallelism for similarity calculation and pipeline parallelism for K-selection. A filter is added between the above two operators to bridge their throughput mismatch and lower the resource requirement. The overall architecture is fully pipelined, with computations and communications perfectly overlapped to minimize latency and maximize throughput.

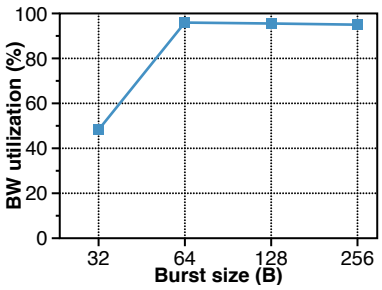


Figure 6: Bandwidth utilization of a single HBM channel with different burst sizes. The utilization is over 90% when the burst size is not smaller than 64 bytes.

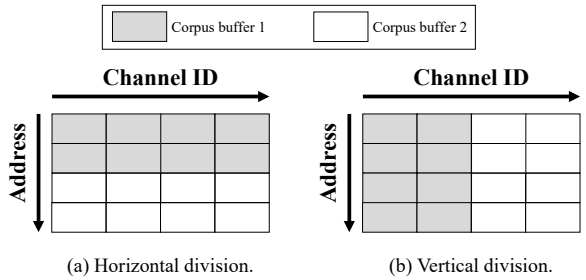


Figure 7: HBM can be divided into two buffers in two ways. (a) Horizontal division: divide buffers based on address, which preserves all memory channels and maximum memory bandwidth for each buffer. (b) Vertical division: divide buffers based on channel ID, which halves the number of available channels and the memory bandwidth for each buffer.

3.1.1 Embedding Compaction to Maximize Single-channel Performance

The bandwidth utilization of a single HBM channel is affected by two factors: *access pattern* (sequential or random access) and *burst size* (the number of bytes in a memory transaction). Given the nature of brute-force KNN search, both corpus scanning and corpus update perform sequential access, which is more efficient than random access. We show in Figure 6 the bandwidth utilization of a single HBM channel in sequential access over various burst sizes. The result reveals that, to achieve bandwidth utilization of over 90%, an ideal burst size should be not smaller than 64 bytes and be a multiple of the channel width of 32 bytes.

However, the size of embeddings could be smaller than 64 bytes, especially for those generated by quantization-aware training [30, 31, 37]. It could also be not a multiple of the channel width. To bridge the mismatch between the ideal burst size requirement and the realistic embedding size, we compact one or multiple embeddings into a burst, whose size might not be exactly a multiple of the embedding size, leaving some unused bytes in a burst. To minimize the waste, we choose an ideal burst size with minimal unused bytes.

3.1.2 Horizontal HBM Division to Maximize Multi-channel Parallelism

To support the online corpus update, the corpus manager partitions HBM into two corpus buffers: a runtime buffer to store the latest corpus and serve queries, and an update buffer reserved for update. Upon receiving a new corpus from the host, the corpus manager stores it into the update buffer, and then switches the EBR pipeline to scan corpus from that buffer for new queries. In this way, the runtime buffer and update buffer switch roles after each update.

HBM can be partitioned into two corpus buffers in two ways, horizontally or vertically, as shown in Figure 7. The horizontal division is chosen, as it keeps all the available HBM channels and thus the maximum memory bandwidth

for each buffer, while the vertical division loses half channels and thus half memory bandwidth for each buffer.

During corpus update, the HBM write caused by update and HBM read caused by query, may contend for HBM memory bandwidth with horizontal division. Such contention is negligible. Considering that the realistic HBM bandwidth is 414 GB/s (90% utilization of a typical 460 GB/s HBM), and corpus update is bounded by the PCIe Gen3 x16 bandwidth (16 GB/s), the update (HBM write) throughput over the total HBM throughput is less than 4%. Moreover, given that corpus update happens much less frequently than query, update can be further throttled to minimize its impact to query. Other update methods will be discussed in §7.

3.2 Similarity Calculation

Similarity calculation receives multiple item embeddings from multiple HBM channels simultaneously. In order to match the bandwidth of HBM, we apply data parallelism in similarity calculation, where multiple scoring units (SU) are instantiated to work in parallel, and each performs similarity calculation, e.g., inner product, between a separate item embedding and the given query embedding. The number of parallel SUs required is the product of the total number of HBM channels and the maximum number of item embeddings inside a channel width, which may contain more than one item embedding due to the embedding compaction (§3.1.1).

3.3 K-selection

There exist multiple different K-selection architectures [27,29, 44], suitable for different scenarios. In the context of recommendation systems, the value of k is from a few thousand to dozens of thousands in realistic EBR [16,25], so K-selection in FAERY aims to achieve both high performance and high scalability in supporting a large value of k. To this end, we choose an existing K-selection pipeline [29] based on bottom-up merge sort for the following two reasons.

First, the bottom-up merge sort allows processing input scores in a streaming manner to avoid storing the entire scores before computing. In contrast, some algorithms incapable of streaming processing, e.g., `RadixSelect` [36], inevitably need external memory to store the entire scores of a large size. Leveraging external memory to cache the scores should be avoided, as it will not only reduce the available storage space for the corpus, but also interfere with the performance of corpus scanning due to bandwidth contention.

Second, pipeline parallelism within K-selection is compute-efficient and scalable, e.g., the chosen K-selection pipeline [29] requires only $O(logk)$ comparators. In contrast, some data-parallel K-selection architectures [27, 44] use a large number of parallel comparators to process a batch of input scores at a time. The number of parallel comparators required by this method is $O(p*k)$, where p is the batch size

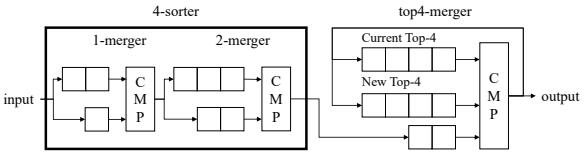


Figure 8: An example of the 4-selection pipeline in [29].

of input scores. Such design is not scalable, especially for k in the order of thousands.

Pipeline parallelism within K-selection. Figure 8 illustrates a K-selection pipeline where $k = 4$. The K-selection pipeline in [29] contains a series of *i-mergers* and a final *topk-merger*. An *i-merger* merges two sorted lists with length i into a sorted list with length $2i$, followed by a *2i-merger* in the pipeline. The pipeline starts at a *1-merger*, and log_2k sequential *i-mergers* form a *k-sorter*. At the end of the pipeline, a *topk-merger* merges the output of the *k-sorter* with the current sorted Top-K to generate a new running Top-K. All modules process data in a streaming manner, and the latency of such a pipeline is $k + log_2k$ clock cycles [29].

The above K-selection pipeline processes one score every clock cycle, which is slower than the throughput of scores generated by similarity calculation with data parallelism. According to the ideal architecture shown in Figure 2, K-selection can simply match the throughput with multiple K-selection pipelines, i.e., instantiating multiple K-selection pipelines in parallel, each processing different scores, followed by a merger at the end to get the final Top-K from multi-channel sorted Top-K. However, a single K-selection pipeline is much more resource-hungry than a single scoring unit. Instantiating multiple K-selection pipelines to match the throughput of the multi-channel similarity calculation is not resource-efficient, especially when supporting a large k and a large batch size. Based on an important observation on the K-selection pipeline, we address the throughput mismatch problem in a resource-efficient way by introducing a new operator: filter (§3.4).

3.4 Filter

The K-selection pipeline maintains inside a **running Top-K** (e.g., the *current Top-4* in Figure 8), which continuously updates the Top-K for all the past scores until the current point. We observe that, if the input score to K-selection is not greater than the minimum score of the running Top-K, the input won't change the internal running Top-K and thus can be dropped. Based on this observation, we design a filter to early drop non-Top-K scores, which significantly reduces the number of scores sent to K-selection. Figure 9 shows the throughput model of FAERY, where corpus scanning and similarity calculation are designed with data parallelism to match the HBM throughput, K-selection only provides a single pipeline to save resources, and the filter bridges the throughput mis-

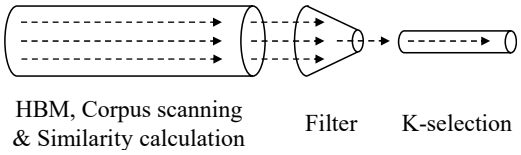


Figure 9: Throughput model of FAERY. Corpus scanning and similarity calculation are designed to fully match the HBM bandwidth, followed by a filter to early drop most of scores generated by similarity calculation, thus significantly lowering the throughput requirement of K-selection.

match between the multi-channel similarity calculation and the single-channel K-selection.

Let x ($x > 1$) denote the number of scores generated by similarity calculation per clock cycle. The throughput of similarity calculation is x scores per clock cycle, and the throughput of K-selection is one score per clock cycle, so that the throughput mismatch is $(x-1)/x$. We define filtering efficiency as the number of scores (m) dropped by the filter over the total number of scores (n), i.e., m/n. As long as $m/n \geq (x-1)/x$, the design will work well without performance degradation.

In practice, the recall ratio of EBR (the ratio of the retrieved items to the total items, i.e., $k:n$) is usually very low, e.g., $1:1000$. The majority of scores will be early dropped by the filter, and the filtering efficiency will be high enough to bridge the throughput gap. We analyze the average filtering efficiency as follows.

Filtering efficiency. Given that $n >> k$ in practice, we can derive the filtering efficiency using a simplified model. Assuming the input scores follow a random distribution, and the running Top-K values are already generated from all the previous scores when the dropping decision for a score is made, the probability of the i^{th} ($i > k$) score dropped by the filter follows

$$p(i) = \frac{i-k}{i}. \tag{2}$$

The expected number of scores dropped by the filter follows

$$m = \sum_{i=k+1}^{n} p(i) = \sum_{i=k+1}^{n} \frac{i-k}{i}. \tag{3}$$

As a result, the average filtering efficiency is

$$\begin{aligned} e &= \frac{m}{n} = \frac{\sum_{i=k+1}^{n} \frac{i-k}{i}}{n} \\ &= 1 - \frac{k}{n} - \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i} \\ &> 1 - \frac{k}{n} - \frac{k}{n} ln(n). \end{aligned} \tag{4}$$

Given a typical setting in practice where $k = 1024$, $n = 10^6$, the filtering efficiency is larger than 98%. In our implementation (§5), x is 4, and the throughput mismatch is $3/4 = 75\%$. This shows that the filtering efficiency is much higher than the throughput mismatch in practice, and thus the filter enables K-selection to match the throughput of similarity calculation with just one pipeline, reducing resource consumption.

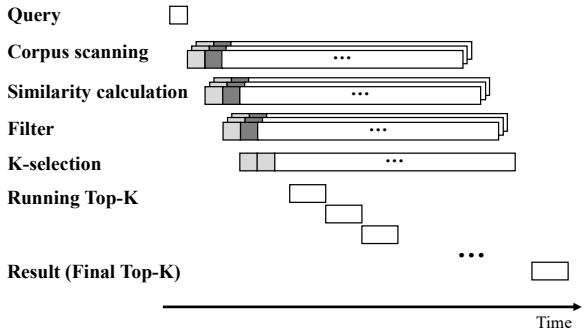


Figure 10: The perfect overlap of data streams in FAERY.

Buffer to absorb bursts. Although the filter balances good performance and low resource cost in practice, it can fail to drop any scores in the worst case when all the input scores are sorted ascendingly. Since the item embeddings are stored randomly in HBM, the probability that such worst case happens is very low. However, we do have a buffer in the filter to absorb two types of temporal bursts. The initial burst is built up while the filter is processing the first y scores of every new query, when the drop probability ($p(i)$, $i < y$) of score i is lower than the throughput mismatch $(x-1)/x$ between similarity calculation and K-selection. Based on Equation 2, y is $(k*x)$. The other type of burst is occasional score sequences in which all scores are larger than the minimum of the running Top-K. The size of this burst is variable but should be small given the increasing drop probability shown in Equation 2.

3.5 Perfect Overlap of FAERY Data Streams

As described in the above sections, all operators work in a streaming manner, i.e., all operators start processing as soon as the data begin to stream in, and the communications between operators are perfectly overlapped with computations. As a result, the data streams in this architecture exhibit a perfect overlap, as shown in Figure 10. Upon receiving a query, the corpus manager starts corpus scanning and gets a multi-stream of embeddings from 32 HBM channels, followed by similarity calculation and filter streams in the subsequent cycles. The filter operator early drops most of scores, so that a single stream of scores is sent to K-selection. As scores begin to stream into K-selection, the running Top-K is updated, and it is output as the final Top-K result soon after the last score is injected into the K-selection pipeline.

Batch is supported in FAERY in the same way as the ideal architecture (Figure 3). The data streams of multiple queries start at the same point. Therefore, the latency remains the same, and the throughput scales linearly with the batch size.

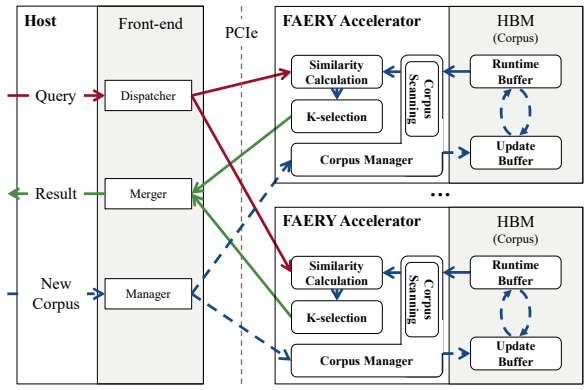


Figure 11: FAERY server architecture. A FAERY server hosts multiple FAERY accelerators to increase either the query throughput or the supported corpus size. A software front-end on the host CPU dispatches user queries to multiple accelerators, merges retrieval results from them, and updates the corpus on the fly.

4 FAERY Server

Figure 11 presents the FAERY server architecture, which includes a software front-end on the host CPU and multiple FAERY accelerators inserted in server PCIe slots.

Multiple FAERY accelerators can work together to enhance the capabilities of a single accelerator in two modes: *replication* and *sharding*. In the *replication* mode, these accelerators store separate replicas of the same corpus and serve different queries simultaneously to increase the query throughput. In the *sharding* mode, multiple accelerators store different shards of the same corpus and serve the same query simultaneously to increase the supported corpus size. The front-end running in the host CPU is responsible for query dispatching via a dispatcher module and result merging via a merger module in the above two modes.

When there is a new corpus received by the server, a manager module in the software front-end handles this update request. It determines whether corpus replicating or sharding is needed based on the working mode listed above, and then sends the corpus replicas (or shards) to the corresponding accelerators via PCIe. The corpus manager in each FAERY accelerator stores the update corpus in the update buffer and switch buffer roles as described in §3.1.2.

5 Implementation

We build a fully functional prototype of FAERY using FPGAs. The FPGA accelerator is built with Xilinx VU35P FPGA [4], which contains an HBM of 8 GB capacity, 32 memory channels, and 460 GB/s bandwidth. We implement the FAERY pipeline described in Figure 5 using the hardware programming language SystemVerilog. In the following part, we discuss several implementation details using this FPGA with a typical setting: One embedding contains 128 elements of 2 bytes each (i.e., the embedding size is 256 bytes), k is 1024, and the prototype runs at a clock frequency of 400 MHz, which matches the HBM bandwidth. An ASIC implementation of FAERY with the same HBM bandwidth but higher clock frequency (e.g., 1 GHz), could not provide significant performance improvement, as the end-to-end performance is mainly determined by the HBM bandwidth.

Corpus manager. Since the embedding size is 256 bytes, the burst size can be set to 256 bytes based on the embedding compaction strategy, resulting in no waste on both storage space and read bandwidth. Based on the measurement, the achievable HBM bandwidth is 414 GB/s, with 90% utilization of the theoretical upper bound of 460 GB/s. Given that the HBM has 32 memory channels of 32-byte width, the corpus scanning reads 1024 ($32*32=1024$) bytes from HBM every clock cycle, almost catching up with the HBM bandwidth at 400 MHz ($1024*400/1000=409.6$ GB/s), and outputs 4 ($1024/256=4$) embeddings per clock cycle on average. To support online corpus update, horizontal division keeps half of the 8 GB HBM space (i.e., 4 GB) for the runtime corpus, which supports up to 16M item embeddings in a single FPGA.

Similarity calculation. To match the throughput of 32 parallel HBM channels, similarity calculation is implemented with 32-channel SUs in parallel. Each SU performs inner product calculation, which consists of three stages. The first stage performs element-wise multiplications between the item and the query. Given that an HBM channel width (32 bytes) contains 16 elements (each 2 bytes) of an embedding, it requires 16 parallel multipliers in this stage to sustain the HBM channel bandwidth. In the second stage, it conducts a summation of the results in the first stage with an accumulation tree, which has $log_2 16=4$ layers. The summation result is finally added to the computing score in the last stage. Therefore, the latency of similarity calculation is 6 ($1+4+1=6$) cycles, and the throughput of similarity calculation with 32 parallel SUs (i.e., 4 scores per clock cycle) matches exactly the throughput of corpus scanning (i.e., 4 item embeddings per clock cycle).

K-selection. K-selection is implemented based on an existing pipeline [29], whose latency is $k+log_2 k$ clock cycles. For $k=1024$, the latency is 1034 cycles. This fully pipelined K-selection can process one score per clock cycle. Different from the ideal architecture, a single K-selection pipeline is required in FAERY, with the filter to bridge the throughput mismatch between similarity calculation and K-selection.

Filter. Since similarity calculation generates four scores per cycle, while K-selection only processes one score per cycle, the filter must drop at least 3/4 of the scores on average to bridge their speed gap. Based on the analysis in §3.4, the filtering efficiency in this setting is higher than 98% and thus greater than 3/4. To absorb bursts, the filter buffer is set to store at most 8192 ($2*k*x$, where $x=4$ and $k=1024$) scores,

	Per-query resources	Common resources
LUT	7.31%	11.05%
FF	6.98%	14.78%
BRAM	13.05%	10.66%
DSP	8.6%	0.07%

Table 2: Breakdown of FAERY resource consumption (batch size = 1). Per-query resources increase linearly with the batch size, while common resources remain unchanged.

slightly larger than the initial burst size ($k * x$) derived in §3.4 to reduce approximation error in the analysis. This buffer is implemented with only 8 Block RAMs (BRAMs), consuming less than 0.2% of the total FPGA memory resources. With both the high filtering efficiency and the sufficient buffer, the filter works well to bridge the throughput mismatch and to absorb temporal bursts, and we observe no performance loss with the above setting. Compared with the ideal architecture shown in Figure 2, FAERY with the filter and a single K-selection pipeline, can save 32% on-chip memories and 27% compute resources by eliminating the other three K-selection pipelines and a four-port merger [35] per query compute pipeline.

Batch support. FAERY supports batch queries as described in Figure 3. Despite the performance advantages, the resource requirements of batch queries increase with the batch size. As a result, the maximum batch size supported in our prototype is determined by the available resources in the Xilinx VU35P FPGA. The resources are consumed by two types of components: per-query compute pipelines (similarity calculation, K-selection, and filter) exclusive for each query, and common modules (corpus manager and PCIe DMA) shared among batch queries. Table 2 breaks down the resource consumption of a FAERY accelerator with the batch size of 1 into per-query resources and common resources. Based on this result, the upper bound of the batch size is 6 in the Xilinx VU35P FPGA. However, this FPGA chip is composed of multiple dies, so that timing closure is challenging when the resource utilization is high or cross-die routing is congested. We end up with an implementation with a batch size of 3, to balance good batch performance and easy timing closure.

6 Evaluation

We evaluate the performance of the FAERY implementation, and compare it with CPU- and GPU-based EBR, respectively. Our results reveal that:

- FAERY approaches the optimal query latency, and achieves 98.09×-118.99× and 1.85×-18.81× lower latency than CPU- and GPU-based EBR, respectively. The degraded FAERY with the same memory bandwidth of GPU still achieves 1.21×-12.27× lower latency than GPU-based EBR.
- In terms of latency-bounded ($\leq$ 10 ms) throughput, FAERY and the degraded FAERY outperform GPU-based EBR by 1.33×-6.58× and 0.87×-4.29×, respectively, while CPU-based EBR fails to meet the 10 ms latency target.
- FAERY achieves 1.66×-8.20× higher energy efficiency and 1.31×-6.46× higher cost efficiency than GPU-based EBR.
- A FAERY server with two accelerators provides 2× higher query throughput in the replication mode, and 2× higher corpus capacity in the sharding mode with less than 1.1% increase in latency.

6.1 Experiment Setup

Baseline. We compare FAERY with Faiss [23], an open-source similarity search library that supports both CPU and GPU. The K-selection implementation in Faiss GPU is `WarpSelect`, a heap-based algorithm using on-chip memory, as shown in Figure 4a, denoted as GPU-o. We further replace the Faiss K-selection implementation with an algorithm using external memory, as shown in Figure 4b, denoted as GPU-e. We choose `RadixSelect` implemented in [33], which reports the best performance when k is greater than 512, compared to other algorithms. Both GPU-o and GPU-e use `fp16` for embeddings and `fp32` for scores.

Platforms. FAERY is evaluated on a server with two 8-core Intel Xeon Silver 4110 CPUs. CPU-based EBR is evaluated on a server with two 16-core Xeon Gold 5218 CPUs and 192 GB memory. We choose Nvidia Tesla T4 GPU [2] in GPU-based EBR, as the T4 GPU shares a similar cost to the Xilinx VU35P FPGA (cost comparison will be discussed in §6.2.4). The CUDA version is 11.2 and the Tensor Core acceleration is enabled. T4 GPU is equipped with 16 GB GDDR6 of 300 GB/s bandwidth. To bridge the difference of memory bandwidth between FPGA (460 GB/s) and GPU (300 GB/s), we also evaluate a degraded FAERY, denoted as FAERY-d, by throttling its HBM bandwidth to 300 GB/s.

Corpus. We use the synthetic corpus, with randomly generated 128-dimensional item embeddings of 2 bytes each dimension, and retrieve $k = 1024$ items for each query. We use synthetic random corpora to verify the generality of FAERY, which by design, is not sensitive to any specific workload.

In the following, we first evaluate the performance of a single accelerator (§6.2). Many important applications contain a moderate corpus. For example, the YouTube video corpus contains tens of millions of items [40], and the Google play application corpus contains one million items [11]. The corpus of these applications could fit into the HBM of a single card based on the current FAERY implementation (§5). Then, we show the performance of a FAERY server with two accelerators (§6.3) to demonstrate FAERY's capability in supporting either higher query throughput or a larger corpus by adding cards.

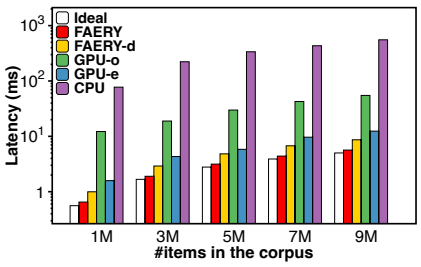


Figure 12: Query latency comparison among different EBR architectures (batch size = 1, latency is in log scale).

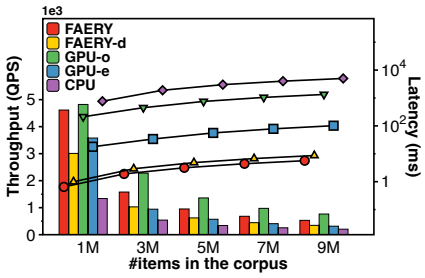


Figure 13: Query throughput comparison among different EBR architectures (Corresponding latency is also shown).

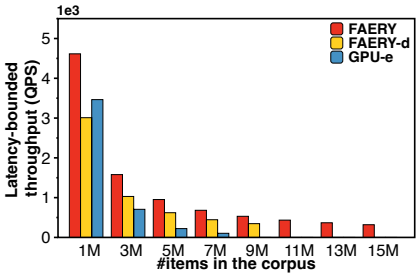


Figure 14: Comparison of latency-bounded throughput, where CPU and GPU-o fail to meet the latency target ($\leq$ 10 ms), and thus are not shown.

6.2 Single-accelerator Performance

6.2.1 Latency

We compare the query latency among different EBR architectures in Figure 12. Average latency is used as the metric, as latency distribution in each of these architectures doesn't show significant variance due to the deterministic execution flow of KNN. The query latency of the ideal architecture is calculated based on Equation 1, where S is $N * 256$ bytes, N is the number of items in the corpus, B is the maximum HBM bandwidth 460 GB/s, and C is the FAERY pipeline latency 2.6 us. The query latency of FAERY approaches the optimal latency of the ideal architecture, with only 1.13×-1.16× increases, which results from non-full ($\sim$ 90%) memory bandwidth utilization as measured in Figure 6. Both FAERY and FAERY-d consistently outperform CPU and GPU in query latency with different corpus sizes. Compared with CPU, FAERY significantly reduces the average latency (98.09×-118.99× lower) due to its high memory bandwidth and appropriate parallelism paradigms for different operators. Compared with GPU, FAERY achieves 9.48×-18.81× and 1.85×-2.44× lower latency than GPU-o and GPU-e, respectively. Even if we degrade the FAERY memory bandwidth to that of GPU T4 (300 GB/s), FAERY-d also achieves 6.18×-12.27× and 1.21×-1.59× lower latency than GPU-o and GPU-e, respectively. This verifies that even with the same memory bandwidth, FAERY-d still outperforms GPU-based EBR, because the poor pipeline support of GPU leads to a significant increase of the second part (C) in Equation 1, as detailed in §2.3.2.

6.2.2 Throughput

We compare the maximum throughput and its corresponding latency among different EBR architectures in Figure 13. Batch queries are used in all architectures to achieve the maximum throughput. Both FAERY and FAERY-d are evaluated with the batch size of 3, the same as that in the implementation. Although the throughput of CPU- and GPU-based EBR systems can be improved by increasing the batch size, we only show the results with the batch size up to 1024, because further increasing the batch size leads to marginal improvement. GPU-o consistently outperforms FAERY in throughput by 1.04×-1.44×, and FAERY-d by 1.60×-2.21×, with a large batch size but a much higher query latency (ranging from 212 ms to 1339 ms with different corpus sizes). In contrast, both FAERY and FAERY-d keep low query latency as that of batch size 1 when increasing the batch size. The throughput of GPU-e does not increase significantly with larger batch sizes, due to heavy contention on external memory bandwidth in K-selection among multiple queries. As a result, GPU-e achieves only 59%-78% (91%-119%) of the FAERY (FAERY-d) throughput, but has a much higher latency (ranging from 18 ms to 102 ms). FAERY outperforms CPU in throughput by 2.60×-3.45× even when the CPU-based EBR runs with a large batch size. Moreover, CPU suffers form the worst latency.

6.2.3 Latency-bounded Throughput

Latency-bounded throughput is a critical metric for EBR, as retrieval is a typical real-time service with strict requirements on the response time. For example, the response time is within 10 ms in the Taobao production retrieval [15, 25], and the query serving time of the entire recommendation pipeline (retrieval + ranking) is on the order of 10 ms in the Google application recommendation [11]. In this paper, we set the upper bound of the retrieval latency to 10 ms, and compare the latency-bounded throughput among different EBR architectures.

Since CPU and GPU-o fail to meet the latency target in any condition, we only compare FAERY and GPU-e in Figure 14. The latency target prevents GPU-e from using a large batch size, which increases per-query latency significantly due to the contention on memory bandwidth. In contrast, FAERY follows the ideal architecture for batch queries, maintaining constantly low latency when increasing the batch size, as discussed in §2.2. When the number of items in the corpus ranges from 1M to 7M, FAERY achieves 1.33×-6.58×

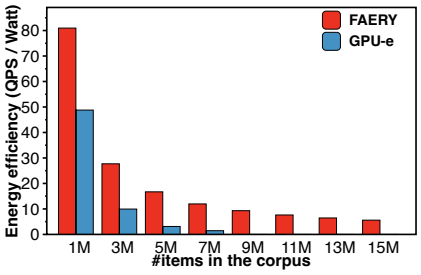


Figure 15: Comparison of energy efficiency among different EBR accelerators.

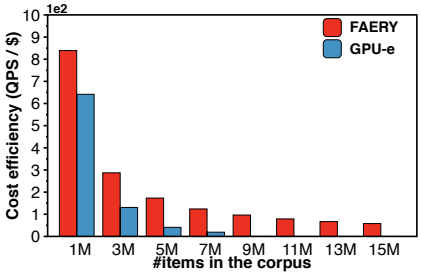


Figure 16: Comparison of cost efficiency among different EBR accelerators.

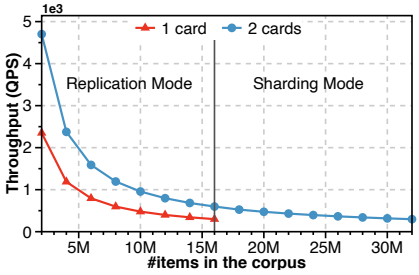


Figure 17: Throughput of a FAERY server with two cards.

higher latency-bounded throughput than GPU-e. However, FAERY-d achieves only 87% of the GPU-e latency-bounded throughput with the small corpus size of 1M, as GPU-e can leverage a large batch size (64 in GPU-e vs. 3 in FAERY-d) to boost the throughput with moderate memory bandwidth contention when the corpus size is small. As the corpus size increases from 3M to 7M items, FAERY-d exhibits its advantages in latency-bounded throughput and achieves 1.46×-4.29× higher latency-bounded throughput than GPU-e. When the number of items is larger than 7M, GPU-e fails to meet the latency target in any batch size, while FAERY-d can increase the corpus size until 9M items under latency target, and FAERY supports up to 15M items.

6.2.4 Energy & Cost Efficiency

The GPU and FPGA used in the evaluation have different architectural advantages and disadvantages, e.g., the GPU has lower memory bandwidth (300 GB/s vs. 460 GB/s), but much higher computing power (130 TOPS vs. 18.6 TOPS for INT8) than the FPGA. In addition to using the degraded FAERY with 300 GB/s memory bandwidth in a direct comparison between FPGA and GPU in terms of latency and throughput, we consider both energy efficiency (performance per watt) and cost efficiency (performance per dollar), as yet another fair metrics to compare the efficiency between totally different hardware architectures. We use the latency-bounded throughput measured in Figure 14 as the performance reference.

Energy efficiency. Based on the measurement, FAERY is 57 Watt and GPU-e is 71 Watt during serving. The above power consumption does not vary significantly with different corpus sizes. Given these power consumption and throughput data, Figure 15 shows the result of energy efficiency (QPS/Watt), where FAERY consistently outperforms GPU-e with 1.66×-8.20× higher energy efficiency.

Cost efficiency. As the concrete cost numbers are confidential, we normalize the costs of GPU, FPGA, and server used in the evaluation to 1, 1.1, and 4.4, respectively. With these cost units, the normalized costs of the FAERY and GPU servers are 5.5 (=1.1+4.4) and 5.4 (=1+4.4), respectively. Based on these normalized costs and the latency-bounded throughput data, Figure 16 shows the result of cost efficiency (i.e., QPS/(cost unit)), where FAERY provides 1.31×-6.46× higher cost efficiency than GPU-e.

6.2.5 Summary

Table 3 summarizes the EBR performance comparison among different processors, and reveals that each processor has its unique advantages for EBR. FAERY, an FPGA-based EBR, achieves the lowest latency, the highest latency-bounded throughput, and the highest energy and cost efficiency compared with CPU and GPU. Compared with CPU, FPGA's performance gain results from the high memory bandwidth provided by HBM and massive programmable compute elements to enable appropriate parallelism paradigms and batch processing. FPGA outperforms GPU due to the fully pipelined design with perfectly overlapping communications with computations of operators, and a programmable architecture that supports efficient K-selection. All these advantages make FAERY not only approach the optimal latency, but also achieve linear-scaling throughput when increasing the batch size. CPU-based EBR supports the largest corpus size, thanks to the large capacity of CPU DDR memory. GPU-based EBR achieves the highest raw throughput without latency bound with a very large batch size, thanks to its massive compute cores.

6.3 Multi-accelerator Performance

We evaluate a FAERY server with two accelerators. Figure 17 shows the aggregate query throughput with different corpus sizes. When the corpus can fit into a single card (i.e., the number of items is not larger than 16M), we replicate the corpus in the two cards to double the query throughput, as shown in the left part of Figure 17. When the corpus size is larger than the memory capacity of a single card, we evenly shard the corpus between the two cards, and thus the supported corpus size is extended up to 32M items, i.e., 2× the HBM capacity of a single card, as shown in the right part of Figure 17. In the

	Corpus size in bytes	Normalized latency	Normalized throughput	Normalized latency-bounded throughput (< 10 ms)	Normalized energy efficiency	Normalized cost efficiency
CPU	> 100 GB	98.09-118.99	0.290-0.385	-	-	-
GPU	16-80 GB	1.85-18.81	0.593-1.440	0.152-0.752	0.122-0.602	0.155-0.763
FPGA (FAERY-d)	8-32 GB	1.53	0.652	0.652	-	-
FPGA (FAERY)	8-32 GB	1	1	1	1	1

Table 3: Summary of performance comparison among different EBR processors.

sharding mode, the software front-end in CPU has to merge the two Top-K results from the two cards and yield the final Top-K, introducing an extra latency of less than 15 us, i.e., 1.1% of the total query latency.

7 Discussion

System lessons. While we focus on FPGA-accelerated EBR in this paper, we believe FPGA is a promising choice for not only EBR acceleration in specific, but also domain specific accelerator (DSA) in general. First, FPGAs are readily available for DSA in several hyper-scale cloud providers [10,14,41,45]. Second, FPGAs are inherently capable of faithfully implementing DSA systems such as FAERY, MicroRec [22], and Tiara [41]. These systems are memory and compute bounded, so they can benefit from customized parallelism and pipelining with optimized memory accesses provided by FPGAs.

Most FPGA-based architectures can be baked into custom ASICs for higher performance and efficiency. In FAERY, the query latency and throughput are mainly limited by the memory bandwidth, so an ASIC implementation with the same memory bandwidth would not significantly improve the performance. However, an ASIC version of FAERY can achieve higher energy efficiency. Nonetheless, it will require a significant volume to amortize the high non-recurring engineering (NRE) cost for higher cost efficiency.

Online update. The online update approach described in §3.1 minimizes the degradation of the total query throughput (QPS) during the update, by taking half of the HBM memory in each card as update buffer. We further note that there are other ways for online update from a distributed system perspective. In a typical production EBR system, there are multiple corpus replicas distributed across multiple FAERY servers for reliability and load balancing purposes. The online update in this case can be performed by taking off one replica at a time for updating while keeping the others online. This approach may achieve higher memory utilization, but experience higher update time and lower QPS than our update approach during the update process.

Support new models. In addition to the online corpus update, FAERY is able to change the pipeline structure on the fly to adapt to new models. Given the relatively stable EBR pipeline structure, including corpus scanning, similarity calculation, and K-selection, we are able to use the same hardware code to support different EBR pipeline variants with just different parameters (e.g., embedding size, data type, k). When a new model requires a change of the pipeline structure, we can simply change parameters in the code, generate a hardware image, and then load the image into FPGA on the fly.

Accelerate ANN-based EBR. Although FAERY is designed to accelerate KNN-based EBR, it can be extended to accelerate ANN to achieve higher throughput by sacrificing retrieval accuracy. Indexing-based ANN algorithms, e.g., IVF [34] and HNSW [28], leverage an index layer before corpus scanning to reduce the number of accessed items per query. Quantization-based ANN algorithms, e.g., PQ [21] and OPQ [17], leverage a codebook to compact the corpus. FAERY can support both ANN variants by maintaining the index layer or the cookbook in FPGA on-chip memory. Most of the other operators are the same, and their designs can be shared among KNN- and ANN-based FAERY.

Use FAERY for other services. Although FAERY is a DSA for retrieval in recommendation systems, we believe a similar idea can be applied to vector search in general, which is a fundamental part of many applications [13, 20, 25] that use semantic embedding vectors to represent contents (articles, images, audios, videos, etc.) and perform searches. These applications share a similar data flow to that described in this paper, but their characteristics vary. Interesting future work is to extend FAERY to accelerate a generic vector search service (such as Microsoft Vector search [5] and Google Vertex AI Matching Engine [6]).

8 Related Work

CPU- and GPU-based EBR systems have been discussed in §2. Existing FPGA-based similarity searches [27,44] were not designed for EBR, and thus not suitable. They leveraged massive parallel comparators to perform K-selection, whose resource consumption is unbearable for k being a few thousand in EBR. Moreover, they did not optimize the efficiency of corpus scanning, as they either did not leverage the high bandwidth of HBM [44] or failed to achieve high bandwidth utilization [27]. There are other kinds of work that accelerated specific ANN

algorithms, e.g., HPQ [7] for quantization-based ANN and QuickNN [32] for indexing-based ANN. They are orthogonal to FAERY that focuses on optimizing the entire EBR pipeline as a whole, including corpus scanning, similarity calculation, and K-selection.

9 Conclusion

FAERY is a domain specific accelerator (DSA) for embedding-based retrieval (EBR). The components of FAERY: corpus scanning, similarity calculation, and K-selection are arranged using the appropriate parallel techniques as required by an ideal EBR architecture. As a result, FAERY does not have the shortcomings and performance penalties of existing CPU- and GPU-based EBR approaches. FAERY not only provides both low latency and high throughput compared with CPU-based EBR, but also outperforms GPU-based EBR in terms of latency-bounded throughput.

Acknowledgments

We thank our anonymous reviewers and shepherd Christopher Rossbach for their insightful comments. We also thank Hong Zhang and Lixin Zheng for all technical discussions and valuable comments. This work is supported in part by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), an HKUST-ByteDance Research Project, and the Hong Kong RGC TRS T41-603/20-R, GRF 16213621 and GRF 16215119.

References

[1] Intel memory latency checker (mlc). https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[2] T4 tensor core datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf.

[3] Theoretical maximum memory bandwidth for intel® core™ x-series processors. https://www.intel.com/content/www/us/en/support/articles/000056722/processors/intel-core-processors.html.

[4] Ultrascale+ fpga product tables and product selection guide. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[5] Vector search - microsoft ai lab. https://www.microsoft.com/en-us/ai/ai-lab-vector-search.

[6] Vertex ai matching engine overview. https://cloud.google.com/vertex-ai/docs/matching-engine/overview.

[7] Ameer MS Abdelhadi, Christos-Savvas Bouganis, and George A Constantinides. Accelerated approximate nearest neighbors search through hierarchical product quantization. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019.

[8] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *International conference on similarity search and applications*, 2017.

[9] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 2013.

[10] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[11] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016.

[12] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, 2016.

[13] Miao Fan, Jiacheng Guo, Shuai Zhu, Shuo Miao, Mingming Sun, and Ping Li. Mobius: towards the next generation of query-ad matching in baidu's sponsored search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[15] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. In *Proceedings of the VLDB Endowment*, 2019.

[16] Weihao Gao, Xiangjun Fan, Chong Wang, Jiankai Sun, Kai Jia, Wenzhi Xiao, Ruofan Ding, Xingyan Bin, Hui Yang, and Xiaobing Liu. Deep retrieval: Learning a retrievable structure for large-scale recommendations. In *arXiv preprint arXiv:2007.07203*, 2021.

[17] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. In *IEEE transactions on pattern analysis and machine intelligence*, 2013.

[18] Mihajlo Grbovic and Haibin Cheng. Real-time personalization using embeddings for search ranking at airbnb. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[19] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[20] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

[21] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. In *IEEE transactions on pattern analysis and machine intelligence*, 2010.

[22] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, et al. Microrec: efficient recommendation inference by hardware and data structure solutions. In *Proceedings of Machine Learning and Systems*, 2021.

[23] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. In *arXiv preprint arXiv:1702.08734*, 2017.

[24] Gwangsun Kim, Jiyun Jeong, John Kim, and Mark Stephenson. Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for gpus. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016.

[25] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. Embedding-based product retrieval in taobao search. In *Proceedings of the 27th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2021.

[26] Jianxun Lian, Fuzheng Zhang, Xing Xie, and Guangzhong Sun. Towards better representation learning for personalized news recommendation: a multi-channel deep fusion approach. In *IJCAI*, 2018.

[27] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. Chip-knn: A configurable and high-performance k-nearest neighbors accelerator on cloud fpgas. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020.

[28] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. In *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[29] Naoyuki Matsumoto, Koji Nakano, and Yasuaki Ito. Optimal parallel hardware k-sorter and top k-sorter, with fpga implementations. In *2015 14th International Symposium on Parallel and Distributed Computing*, 2015.

[30] Yuriy Mishchenko, Yusuf Goren, Ming Sun, Chris Beauchene, Spyros Matsoukas, Oleg Rybakov, and Shiv Naga Prasad Vitaladevuni. Low-bit quantization and quantization-aware training for small-footprint keyword spotting. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, 2019.

[31] Hieu Duy Nguyen, Anastasios Alexandridis, and Athanasios Mouchtaris. Quantization aware training with absolute-cosine regularization for automatic speech recognition. In *Interspeech*, 2020.

[32] Reid Pinkham, Shuqing Zeng, and Zhengya Zhang. Quicknn: Memory and performance optimization of kd tree based nearest neighbor search for 3d point clouds. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[33] Anil Shanbhag, Holger Pirk, and Samuel Madden. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[34] Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*, 2003.

[35] Wei Song, Dirk Koch, Mikel Luján, and Jim Garside. Parallel hardware merge sorter. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[36] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[37] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. Degree-quant: Quantization-aware training for graph neural networks. In *arXiv preprint arXiv:2008.05000*, 2020.

[38] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[39] Ruobing Xie, Zhijie Qiu, Jun Rao, Yi Liu, Bo Zhang, and Leyu Lin. Internal and contextual attention network for cold-start multi-channel matching in recommendation. In *IJCAI*, 2020.

[40] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed Chi. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Proceedings of the 13th ACM Conference on Recommender Systems*, 2019.

[41] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[42] Han Zhang, Songlin Wang, Kang Zhang, Zhiling Tang, Yunjiang Jiang, Yun Xiao, Weipeng Yan, and Wen-Yun Yang. Towards personalized and semantic retrieval: An end-to-end solution for e-commerce search via embedding learning. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020.

[43] Heng-Ru Zhang, Fan Min, Zhi-Heng Zhang, and Song Wang. Efficient collaborative filtering recommendations with multi-channel feature vectors. In *International Journal of Machine Learning and Cybernetics*, 2019.

[44] Jialiang Zhang, Soroosh Khoram, and Jing Li. Efficient large-scale approximate nearest neighbor search on opencl fpga. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[45] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. Fpga-accelerated compactions for lsm-based key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[46] Weijie Zhao, Shulong Tan, and Ping Li. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[47] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: a versatile programming framework for pipelined computing on gpu. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

Efficient and Scalable Graph Pattern Mining on GPUs

Xuhao Chen
MIT CSAIL

Arvind
MIT CSAIL

Abstract

Graph Pattern Mining (GPM) extracts higher-order information in a large graph by searching for small patterns of interest. GPM applications are computationally expensive, and thus attractive for GPU acceleration. Unfortunately, due to the complexity of GPM algorithms and parallel hardware, hand optimizing GPM applications suffers programming complexity, while existing GPM frameworks sacrifice efficiency for programmability. Moreover, little work has been done on GPU to scale GPM computation to large problem sizes.

We describe G^{2}Miner, the first GPM framework that runs efficiently on multiple GPUs. G^{2}Miner uses *pattern-aware*, *input-aware* and *architecture-aware* search strategies to achieve high efficiency on GPUs. To simplify programming, it provides a code generator that automatically generates pattern-aware CUDA code. G^{2}Miner flexibly supports both breadth-first search (BFS) and depth-first search (DFS) to maximize memory utilization and generate sufficient parallelism for GPUs. For the scalability of G^{2}Miner, we propose a customized scheduling policy to balance workload among multiple GPUs. Experiments on a V100 GPU show that G^{2}Miner is 5.4× and 7.2× faster than the two state-of-the-art single-GPU systems, Pangolin and PBE, respectively. In the multi-GPU setting, G^{2}Miner achieves linear speedups from 1 to 8 GPUs, for various patterns and data graphs. We also show that G^{2}Miner on a V100 GPU is 48.3× and 15.2× faster than the state-of-the-art CPU-based systems, Peregrine and GraphZero, on a 56-core CPU machine.

1 Introduction

Graph Pattern Mining (GPM) finds subgraphs in a given data graph which match the given pattern(s) (Fig. 1). GPM is a key building block in many domains, e.g., protein function prediction [6, 29, 83], network alignment [62, 76], spam detection [9, 34, 37], chemoinformatics [31, 57, 84], sociometric studies [36, 48], image segmentation [119]. Graph machine learning tasks can also benefit from GPM, including anomaly detection [4, 78], entity resolution [12], community detection [90], role discovery [88] and relational classification [61].

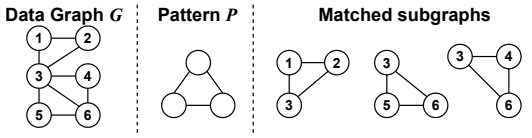


Figure 1: Graph Pattern Mining example. The pattern $\mathcal{P}$ is a triangle, and 3 triangles are found in the data graph $\mathcal{G}$.

GPM is extremely compute intensive, since it searches a space that is exponential in the pattern size. For example, Peregrine [53], a state-of-the-art GPM system on CPU, takes 9 hours to mine the `4-cycle` pattern (see Fig. 3) in the `Friendster` graph on a 56-core CPU machine. GPUs provide much higher compute throughput and memory bandwidth than CPUs, and thus are attractive for GPM acceleration.

However, implementing GPM on GPU efficiently is challenging. This is because it requires sophisticated optimizations by leveraging information in the GPU hardware architecture, the pattern(s) of interest, and the input data graph.

- *Architecture Awareness*: A GPU usually has smaller memory capacity than a CPU and requires more fine-grain data parallelism to be fully utilized. More threads, however, require more memory to accommodate intermediate data! The search order, BFS or DFS, offers a similar tradeoff between memory and parallelism and therefore, GPM on GPU requires careful orchestration of parallelism and memory usage to maximize efficiency. GPUs are also much more sensitive to thread divergence and workload imbalance [18] than CPUs. This necessitates a more sophisticated task-to-hardware mapping for GPU than that for CPU.
- *Pattern Awareness*: State-of-the-art GPM systems on CPU use *pattern aware* search plans that prune the search space using pattern information. This has been shown to be orders-of-magnitude faster than the pattern-oblivious search [53]. This pattern-aware approach has worked well for CPU, but it has not been well explored on GPU. For example, many

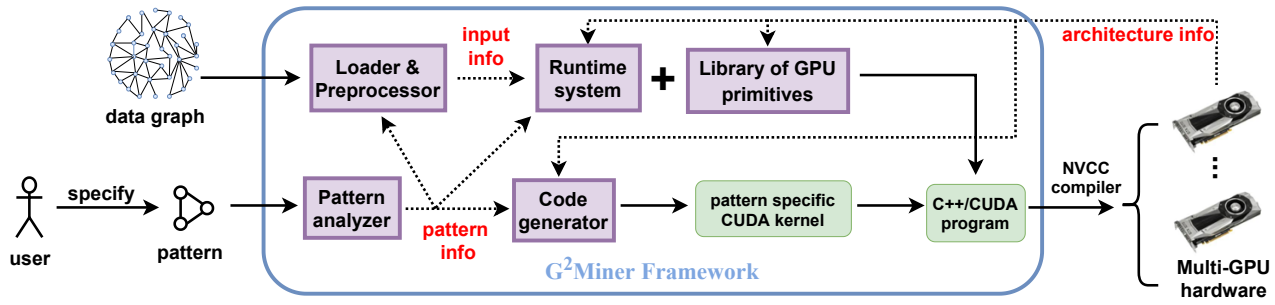


Figure 2: G^{2}Miner system overview. It contains a graph loader, a pattern analyzer, a runtime, a library of GPU primitives and a code generator.

pattern-aware pruning schemes are only effective under DFS exploration, but existing GPU-based systems use BFS, and thus loss these opportunities for pruning.

- *Input Awareness*: Dynamic memory allocation is expensive in GPU [110] and we can avoid it if we can estimate the worst-case memory usage. This can be done by using some meta information such as the maximum degree of the input graph. For labelled graphs, we can use the vertex label distribution of the input graph to get the maximum number of possible patterns, which helps save memory space. In general, input information helps make better tradeoff among work efficiency, parallelism and memory consumption.

Given this complexity, the design goal of our GPM framework on GPUs involves the following considerations:

- **Efficiency:** To achieve high efficiency on GPU, a GPM system must be highly optimized with awareness of the pattern, the input and the hardware architecture. There is no prior system, neither on CPUs nor on GPUs, that considers all three aspects together. This asks for a holistic solution that incorporates sophisticated optimizations systematically.
- **Ease of programming:** Writing efficient GPM code on GPUs is particularly difficult for domain users, who may not be parallel programming experts. Thus, hiding GPU programming complexity is essential for system usability.
- **Scalability:** The skewness in power-law graphs causes load imbalance. This problem is exacerbated for DFS-based GPM algorithms, because accesses to neighbors are multiple hops away. Hence, we need effective task scheduling and distribution policies to scale to multiple GPUs.

	General	CPU	GPU	Multi-GPU	Order	Code Gen
EmptyHeaded [2]		✓			DFS	✓
Graphflow [7,55,75]		✓			DFS	
GraphZero [73,74]		✓			DFS	✓
GraphPi [93]		✓			DFS	✓
Peregrine [53]	✓	✓			DFS	
Pangolin [25,26]	✓	✓	✓		BFS	
PBE [42,43]			✓		BFS	
G^{2}Miner	✓		✓	✓	both	✓

Table 1: Comparison of state-of-the-art GPM systems, in terms of support for generality of the programming model, hardware platforms (CPU/GPU/multi-GPU), search orders, and code generation.

We propose G^{2}Miner to overcome these challenges. Table 1 compares G^{2}Miner to the state-of-the-art systems, including those that solve only the *subgraph matching* problem, which is a subset of the GPM problem. In Table 1, subgraph matching systems include EmptyHeaded, Graphflow, GraphZero, GraphPi and PBE, while Peregrine, Pangolin and G^{2}Miner are general GPM systems. Much of the prior work focuses on CPU, and uses DFS to reduce the memory footprint. GPU-based systems (Pangolin and PBE), on the other hand, use BFS because straightforward DFS implementations on GPU suffer from thread divergence and load imbalance. This, however, limits their efficiency and/or the problem size they can solve. Additionally, G^{2}Miner simplifies GPU programming with automated CUDA code generation, while Pangolin requires users to write CUDA code manually, and PBE is not programmable at all. Last but not least, G^{2}Miner is the only system that scales to multiple GPUs.

Fig. 2 shows the overview of G^{2}Miner. It consists of a graph loader, a pattern analyzer, a runtime system, a library of CUDA primitives and a code generator. The user is only responsible for specifying the pattern(s) of interest using our API (§4). The pattern analyzer does analysis on the pattern and generates a *pattern-specific* search plan, based on which, the code generator (§5) automatically generates pattern-specific CUDA kernels for GPUs. The kernels contain invocations to the device functions defined in the GPU primitive library (§6) which includes efficiently implemented set operations. The generated kernels, the GPU primitive library, and the runtime are compiled together by the NVCC compiler to generate the executable that runs on multi-GPU.

At runtime, the graph loader reads in the data graph, extracts input information (e,g., maximum degree and label distribution) and performs pattern-specific preprocessing on the data graph. The pattern, input and architecture information is fed to the runtime (§7) which heuristically handles GPU memory allocation, data transfer, and multi-GPU scheduling.

This paper makes the following contributions:

- G^{2}Miner is the first pattern-aware, input data-graph-aware and architecture-aware framework for GPM, and it is the first GPM system that automates CUDA code generation for arbitrary patterns to simplify programming.

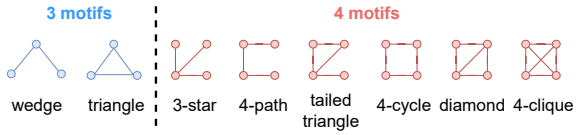


Figure 3: 3-vertex (left) and 4-vertex (right) motifs [26].

- G^{2}Miner is the first multi-GPU framework for GPM and the first GPU-based GPM framework that flexibly supports both BFS and DFS. It uses a novel task scheduling policy to balance workload among GPUs and we show G^{2}Miner performance increases linearly from 1 to 8 V100 GPUs.
- On a V100 GPU, G^{2}Miner is 5.4× faster than *Pangolin*, the only existing GPM system on GPU, and 7.2× faster than *PBE*, the state-of-the-art subgraph matching solver on GPU, thanks to the optimizations enabled in G^{2}Miner (Table 2).
- G^{2}Miner on a V100 GPU is 48.3× and 15.2× faster than state-of-the-art CPU-based GPM system *Peregrine* and subgraph matching system *GraphZero* on a 56-core CPU.

2 Background and Related Work

2.1 Graph Pattern Mining Problems

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be an undirected graph with $\mathcal{V}$ as the vertex and $\mathcal{E}$ as the edge set. Given a vertex $v \in \mathcal{V}$, the neighbor set of v is $\mathcal{N}(v)$, the degree d_v of v is $|\mathcal{N}(v)|$ and Δ is the maximum degree in $\mathcal{G}$. A graph $G'(W,F)$ is said to be a subgraph of $\mathcal{G}$ if $W \subseteq \mathcal{V}$ and $F \subseteq \mathcal{E}$. G' is a *vertex-induced subgraph* of $\mathcal{G}$ if F contains all the edges in E whose endpoints are in W. G' is an *edge-induced subgraph* of $\mathcal{G}$ if W contains all the vertices in $\mathcal{V}$ which are the endpoints of edges in F.

Definition of GPM. Given an undirected graph $\mathcal{G}$ and a set of patterns S_p={P_1, P_2, ...} by the user, GPM finds vertex-induced or edge-induced subgraphs in $\mathcal{G}$ that are isomorphic to any $\mathcal{P}$ in S_p. If the cardinality of S_p is 1, we call it a single-pattern problem. Otherwise, it is a multi-pattern problem. The output of GPM varies in different problems, e.g., the pattern frequency (a.k.a, *support*) or listing all matched subgraphs. The definition of support also varies, e.g., the count of matches or the *domain support* [26] used in FSM. Note that *listing* requires enumerating every subgraph, but *counting* does not. Thus, counting allows more aggressive search-space pruning.

A pattern $\mathcal{P}$ is a small graph that can be defined explicitly or implicitly. An explicit definition specifies the vertices and edges of $\mathcal{P}$, whereas an implicit definition specifies the desired properties of $\mathcal{P}$. For explicit-pattern problems, the solver finds matches of $\mathcal{P}$ in S_p. For implicit-pattern problems, S_p is not known in advance. Therefore, the solver must find the patterns as well as their matches during the search.

GPM requires guarantee for *completeness*, i.e., every match of $\mathcal{P}$ in $\mathcal{G}$ should be found, and often *uniqueness*, i.e., every distinct match should be reported only once [101]. To avoid confusion, we call a vertex in the pattern $\mathcal{P}$ as a *pattern vertex* and denote it as u_i, and a vertex in the data graph $\mathcal{G}$ as a *data vertex* and denote it as v_i. Our work covers the following GPM problems from the literature [26, 33, 101]:

- *Triangle counting* (TC): It counts the number of triangles (Fig. 1), i.e., 3-cliques, in $\mathcal{G}$.
- *k-clique listing* (k-CL): It lists all the k-cliques in $\mathcal{G}$ ($k \geq 3$). A k-clique is a k-vertex graph whose every pair of vertices are connected by an edge.
- *Subgraph listing* (SL). It lists all edge-induced subgraphs of $\mathcal{G}$ that are isomorphic to a pattern $\mathcal{P}$.
- *k-motif counting* (k-MC): It counts the number of occurrences of all possible k-vertex patterns. Each pattern is called a *motif* [11, 77]. Fig. 3 shows all 3-motifs and 4-motifs. This is also an example of a multi-pattern problem because we have to find all the subgraphs that are isomorphic to *any* pattern in a given set of patterns.
- *k-frequent subgraph mining* (k-FSM): Given k and a threshold σ_{min}, this problem considers all patterns with fewer than k edges and lists a pattern $\mathcal{P}$ if the support σ of $\mathcal{P}$ is greater than σ_{min}. This is called a *frequent* pattern. If k is not specified, it is set to ∞, meaning that it is necessary to consider all possible values of k. In k-FSM, vertices in $\mathcal{G}$ have application-specific labels.

For TC and k-CL, vertex-induced and edge-induced subgraphs are the same. SL and FSM find edge-induced subgraphs, while k-MC looks for vertex-induced subgraphs. All problems seek to find explicit pattern(s) except FSM which finds implicit patterns. k-MC and FSM are multi-pattern problems, while the others are single-pattern problems.

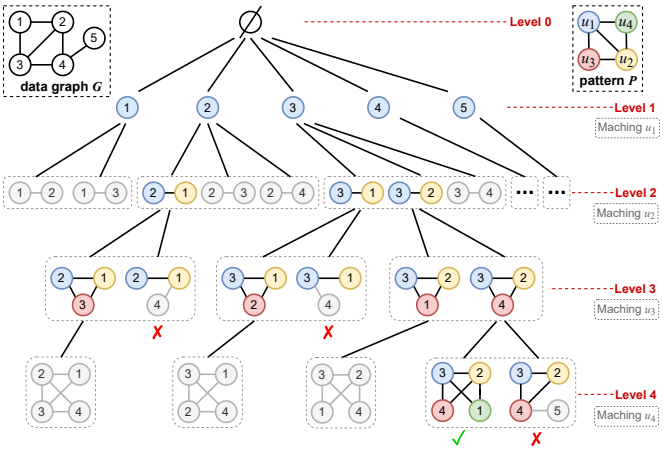


Figure 4: A search tree using vertex extension. Vertex colors (not vertex labels) show the matching between data vertices and pattern vertices. The matching order is $\{u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4\}$. The symmetry order is $\{v_a > v_b, v_c > v_d\}$. Subgraphs in grey are ruled out by symmetry breaking. × shows the unnecessary extensions that are pruned by the matching order. ✓ shows the matched subgraph.

Algorithm 1 Pseudo code for finding `diamond` in DFS order

```
1: for each vertex v1 ∈ V in parallel do                    ▷ match v1 to u1
2:     for each vertex v2 ∈ N(v1) do                        ▷ match v2 to u2
3:         if v2 ≥ v1 then break;                           ▷ symmetry breaking
4:         W ← N(v1) ∩ N(v2);                               ▷ set intersection: buffered in W
5:         for each vertex v3 ∈ W do                        ▷ match v3 to u3
6:             for each vertex v4 ∈ W do                    ▷ match v4 to u4; W is reused
7:                 if v4 ≥ v3 then break;                   ▷ symmetry breaking
8:                 else count ++;                           ▷ do the counting
```

Algorithm 2 Pseudo code for finding Pattern $\mathcal{P}$ in BFS order

```
1: for each level i ∈ [1, P.size] do          ▷ level i from 1 to the pattern size
2:     for each subgraph sg ∈ SL_i in parallel do          ▷ SL_i: subgraph list
3:         for each vertex u ∈ sg do
4:             for each vertex v ∈ N(u) do
5:                 sg' ← sg ∪ v                ▷ vertex extension: add vertex v
6:                 if sg' satisfy P.constraints(i) then
7:                     if i = P.size then count ++;      ▷ leaf: a match found
8:                     else SL_{i+1}.insert(sg')         ▷ go to the next level
```

2.2 Pattern-Aware GPM Algorithms

A GPM problem is a search problem, whose search space is a *subgraph tree* [25, 27] (Fig. 4). Each node in the tree is a subgraph of the data graph $\mathcal{G}$. Subgraphs in level l of the tree have l vertices. The root of the tree (level 0) is an empty subgraph, while the leaves of the tree are potential candidates of matches. A GPM problem can be solved by building this search tree, and checking each leaf if it is isomorphic to the pattern $\mathcal{P}$ using the typical *graph isomorphism test*.

The search tree is built by *vertex extension*: subgraph S_1=(W_1, E_1) can be extended by a single vertex $v \notin W_1$ to obtain subgraph S_2=(W_2, E_2), if v is connected to some vertex in W_1 (i.e., v is in the *neighborhood* of subgraph S_1). When two subgraphs are related in this way, we say that S_2 is a child of S_1. Formally, this can be expressed as W_2=$W_1 \cup \{v\}$ where $v \notin W_1$ and there is an edge $(v,u) \in \mathcal{E}$ for some $u \in W_1$. Similarly, *edge extension* extends a subgraph S_1 with a single edge (u,v), with at least one of the endpoints of the edge is in S_1.

The efficiency of a GPM algorithm depends heavily on how much we can prue the search tree. State-of-the-art GPM frameworks [53, 74] use *pattern-aware* search plans that leverage the properties of the pattern to prune the tree. A pattern-aware search plan consists of a *matching order* and *symmetry order*.

Matching order is a total order that defines how the data vertices are matched to pattern vertices. This order is used to eliminate irrelevant subgraphs on-the-fly. As shown in Fig. 4, to find the `diamond` pattern, we use a matching order among pattern vertices: $\{u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4\}$, meaning that each vertex v_1 added at level 1 is matched to u_1; each vertex v_2 added at level 2 are matched to u_2, and so on. To search for matching candidates, there are connectivity constraints for the data vertices. For example, in `diamond`, since u_3 is connected to both u_1 and u_2, candidate vertices of v_3 must be found in the intersection of v_1 and v_2's neighborhoods, i.e., $v_3 \in \mathcal{N}(v_1) \cap \mathcal{N}(v_2)$. The same constraint should also be applied to v_4. For a given pattern $\mathcal{P}$, there exist multiple valid matching orders. To choose the best performing matching order, prior works [7, 21, 22, 53, 59, 73, 74, 93] have proposed various cost models to predict the performance of matching orders, and choose the one with the highest expected performance.

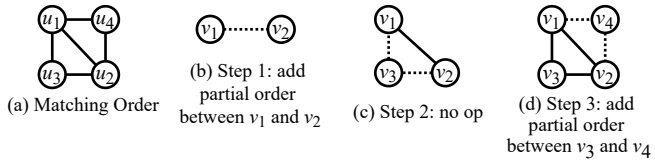
Symmetry order is a partial order enforced among data vertices for *symmetry breaking*, which removes redundant subgraph enumerations (a.k.a *automorphism* [26]), and thus guarantees that any match of $\mathcal{P}$ in $\mathcal{G}$ is found only once. For example, for `diamond`, we enforce that vertices added at level 1 must have larger ids than vertices added at level 2, i.e., $v_1 > v_2$. Thus, in level 2 of the tree in Fig. 4, the subgraph $\{2, 1\}$ is selected to be extended further, but subgraph $\{1, 2\}$ is pruned. Similarly we add a constraint that $v_3 > v_4$. So the symmetry order for `diamond` is $\{v_1 > v_2, v_3 > v_4\}$.

2.3 DFS vs. BFS

Any search order (e.g., BFS, DFS) can be used to explore the search tree, but different search orders come with different work efficiency, parallelism and memory consumption.

Algorithm 1 shows a DFS algorithm to mine the pattern `diamond`. It contains 4 nested `for` loops (Line 1, 2, 5, 6). Each loop corresponds to a data vertex (v_1, v_2, v_3, v_4) that is mapped to a pattern vertex (u_1, u_2, u_3, u_4) in Fig. 5 (a). A buffer W in Line 4 holds intermediate data that is reused multiple times, which avoids redundant computation and thus improves *work efficiency*. The memory footprint contains only four vertices $(v_i, i = 1, 2, 3, 4)$ and W in Line 4 whose size is bounded by Δ. In DFS, every parallel *task* does a DFS walk on the entire subtree rooted at v_1 (Line 1). This is known as *vertex parallelism*. The amount of parallelism is $|\mathcal{V}|$. Another way to parallelize it is *edge parallelism*, in which every task contains the subtree rooted at each edge (say, if we make Line 2 in parallel). The amount of parallelism then is $|\mathcal{E}|$.

The BFS algorithm in Algorithm 2 explores the tree level by level. In each level, it maintains a *subgraph list* that is shared globally among all threads. Each thread takes a subgraph from the subgraph list (Line 2), and extends it to generate its child subgraphs (Line 5). The child subgraphs are inserted into the next-level subgraph list (Line 8). In BFS, each parallel task is a subgraph in the subgraph list of the current level. Since the size of the subgraph list increases exponentially level by level, the amount of parallelism increases rapidly.



(a) Matching Order (b) Step 1: add partial order between v_1 and v_2 (c) Step 2: no op (d) Step 3: add partial order between v_3 and v_4

Figure 5: Generating symmetry order for `diamond` [27].

Although it provides more parallelism than DFS, BFS needs much more memory to accommodate the subgraph list. For example, the BFS-based GPM system Pangolin [26] needs more than 40GB memory to mine the 5-`clique` pattern in a moderate size graph `livejournal`, making it impossible to run in most of off-the-shelf GPUs.

2.4 GPM Systems and Applications

Many existing GPM systems [20, 26, 101, 107, 120] use the BFS order. As they do level-by-level subgraph extension, they generate massive intermediate data and thus are limited to small graphs and patterns. Recently, a few DFS-based GPM systems [25, 33, 53, 74] have been proposed to support larger datasets, but they are all CPU-based. Among all, Pangolin is *the only existing GPM system that supports GPU*. However, limited by the BFS order, Pangolin can only handle small graphs, and it lacks pattern and input awareness.

There also exist subgraph matching systems on CPU [2, 55, 73, 93, 104] and GPU [42, 43, 117]. But they only support a subset of GPM problems and are usually not programmable.

Numerous hand-optimized GPM applications have been developed, including triangle counting [32, 39, 47, 50, 80, 81, 94, 98, 109, 111, 116] , k-clique listing [30] and counting [5, 19, 52, 92], motif counting [3, 67, 70, 82, 89, 95], subgraph listing/matching [13, 14, 17, 46, 54, 59, 60, 65, 68, 72, 85, 86, 91, 96, 97, 103, 105, 108], and FSM [1, 35, 58, 99, 100, 102, 106, 114]. All of them are manually optimized with significant programming effort to achieve high efficiency, which is quite a lot of burden for the domain programmers.

3 Challenges of Efficient GPM on GPU

3.1 GPM vs. Graph Analytics

Similar to graph analytics, GPM algorithms are *irregular* [18] because the control flow and memory accesses are input-data dependent and thus, cannot be predicted statically. This irregularity causes random memory accesses and load imbalance, making it difficult to be efficiently parallelized. Unlike graph analytics that only accesses 1-hop neighbors, GPM requires accessing multi-hop neighbors, which exacerbates the irregularity problem. For example, load imbalance is much worse for DFS-based GPM than graph analytics because each parallel *task* (a DFS walk on the entire sub-tree) is more coarse-grain in GPM. In addition, GPM generates intermediate data during the search (buffer W in DFS or subgraph list in BFS), which consumes extra memory than graph analytics.

3.2 GPM on GPU vs. GPM on CPU

Since the tasks are independent of each other, they are fairly easy to parallelize on CPU, as shown in Algorithm 1 Line 1. But it is not as straightforward on GPU due to GPU's massively parallel model and limited memory capacity.

A GPU often consists of multiple *streaming multiprocessors* (SM). Each SM accommodates multiple vector units. This hardware organization results in a hierarchical parallel model: each CUDA *kernel* includes groups of threads called *cooperative thread arrays* (CTAs) or *thread blocks*. Within each CTA, subgroups of threads called *warps* are executed simultaneously. Thus GPUs, to be fully utilized, require much more hierarchical parallelism than CPUs.

GPUs generally have less memory than CPUs, while BFS-based GPM algorithms consume memory exponential in the pattern size. Using DFS can reduce memory consumption, and also improve work efficiency. Hence, state-of-the-art CPU-targeted GPM frameworks [25, 53, 73, 74, 93] all adopt DFS. However, naively porting the DFS-based CPU algorithms to GPU is not efficient because of the following reasons:

(1) Branch Divergence. In Algorithm 1, each thread takes a vertex v_1 from $\mathcal{V}$ and starts DFS walk rooted by v_1. Since different vertices have different neighborhoods, the threads in a warp may take different paths at the branches, leading to inefficiency on GPU [87]. Branch divergence is much more severe for DFS than BFS due to the multiple nested loops for DFS backtracking that access multi-hop neighborhoods.

(2) Memory Divergence. DFS walk also makes memory accesses more irregular. This causes memory divergence in GPU, i.e., threads in a warp access non-consecutive memory locations. In this case, each load instruction generates multiple (up to the warp size, i.e., 32) memory requests to the memory subsystem, which wastes memory bandwidth, congests on-chip data path [24], and thus results in poor GPU performance.

(3) Load Imbalance. Variance of neighborhood sizes in power-law graphs causes load imbalance. In CPU it is less significant because there are limited number of cores/threads and each core is very powerful. However, GPUs have thousands of lightweight cores and more than ten times the number of active threads. If unbalanced, it would be much more costly since the slowest thread is running on a low-frequency core and thousands of cores are waiting. Load imbalance is also less concerned for BFS, since it does level-by-level extension and at each level the tasks are lightweight, i.e., fine-grained.

Therefore, existing GPU-based GPM systems [26] and subgraph matching systems [42, 43] all use BFS order. This severely limits the graph sizes that they can handle. PBE [42] partitions the data graph to support large graphs, but partitioning introduces cross-partition communication. Note that using *beam search* [71] or bounded DFS does not fully resolve these issues, but loses the benefit of work efficiency of using DFS.

4 G^{2}Miner System Overview and Interface

We propose G^{2}Miner (Fig. 2) to address the challenges in §3. It hides away GPU programming complexity, and takes into

Listing 1: k-Clique Listing (k-CL) user code in G^{2}Miner

```
1 Graph G = loadDataGraph("graph.csr");
2 Pattern p = generateClique(k);
3 list(G, p); // count(G, p) for counting
```

Listing 2: Subgraph Listing (SL) user code in G^{2}Miner

```
4 Pattern p("pattern.el", EdgeInduced);
5 list(G, p);
```

Listing 3: k-Motif Counting (k-MC) user code in G^{2}Miner

```
6 Set<Pattern> patterns = generateAll(k);
7 Map<Pattern,int> result = count(G, patterns);
```

Listing 4: Frequent Subgraph Mining (k-FSM) user code in G^{2}Miner

```
8  Void updateSupport(Subgraph s) {
9    map(s.getPattern(), s.getDomain());
10 }
11 bool patternFilter(Pattern p) {
12   return p.getDomainSupport() >= threshold;
13 }
14 Set<Pattern> patterns = generateAll(k,
15                  EdgeInduced, patternFilter);
16 list(G, patterns, PATTERN_ONLY);
```

account the properties of the pattern, input data graph and hardware architecture to achieve high efficiency on GPU. We first describe how to program in G^{2}Miner in §4.1, and then introduce the system interface for extracting information out of the input, pattern and architecture (§4.2). Lastly we give an overview of the optimizations in §4.3.

4.1 Making Programming Easy

G^{2}Miner provides the same API as state-of-the-art CPU-based systems, e.g., Peregrine and Sandslash, making it friendly to users of CPU frameworks. As shown in Listing 1, to program a k-CL solver in G^{2}Miner, the user specifies the pattern using an utility function `generateClique()` (Line 2), and then call `list()` to do listing or `count()` to do counting. If `count()` is used, it allows the system enable counting-only optimizations (details in §5). To list an arbitrary pattern $\mathcal{P}$ (Listing 2), the user can specify $\mathcal{P}$ using its edgelist (`pattern.el` at Line 4). By default G^{2}Miner finds vertex-induced subgraphs. Since SL requires listing edge-induced subgraphs by definition, the user needs to specify it (`EdgeInduced` at Line 4).

For multi-pattern problems, the user is interested in a set of patterns instead of just one. For k-MC in Listing 3, the patterns can be generated by calling an utility function `generateAll()` (Line 6) or parsing the patterns' edgelists.

Programmability is particularly important for implicit-pattern problems. The user must implement API functions to specify the patterns. For example, for k-FSM in Listing 4, the user chooses to use `domain support` by implementing `updateSupport` (Line 8). To specify the properties that differentiate the interesting patterns with irrelevant patterns, the user must define `patternFilter` (Line 11). As FSM asks for only listing the patterns, we can specify a `PATTERN_ONLY` keyword in `list` to avoid listing the subgraphs (Line 16). If the user wants to customize the output, one can define a `output()` function and pass it to `list`, instead of using `PATTERN_ONLY`. This function defines custom operations on each subgraph of interest, which can also be used to do *early termination* [53] by checking a user-defined condition.

4.2 System Interface

The pattern specified by user API is fed to a *pattern analyzer* to extract useful pattern information. Meanwhile, the GPU hardware information is taken by G^{2}Miner to enable optimizations in the runtime, code generator and GPU primitives. At runtime, the data graph is loaded by a *graph loader* which collects input information and also performs preprocessing.

Pattern Analyzer. The *pattern analyzer* generates: (1) a search plan with a matching order and a symmetry order, which is used by the code generator; (2) reuse opportunities using buffers (e.g., W in Algorithm 1), used by the code generator and the runtime; (3) other important properties of the pattern, e.g., whether the pattern is a clique or `hub-pattern` (§5.4 (2)), used by the runtime and code generator.

The pattern analyzer enumerates all the possible matching orders of $\mathcal{P}$, and uses a cost model to pick the best one. We use the same cost model as GraphZero [73] for fair comparison, but any cost model can be employed by G^{2}Miner. We also use the algorithm in GraphZero to generate a symmetry order: it takes the generated matching order $\mathcal{MO}$ and builds a subgraph incrementally in the order specified by $\mathcal{MO}$. At each step it detects symmetric vertex pairs and adds orders accordingly. For example, for `diamond`, the matching order in Fig. 5 (a) results in the three steps shown in (b), (c) and (d), during which we add partial order $v_2 < v_1$ and $v_4 < v_3$.

Graph Loader and Preprocessor. The data graph $\mathcal{G}$ is loaded by the *graph loader* into the memory in the compressed sparse row (CSR) format. As $\mathcal{G}$ is being loaded, useful input information of the data graph is extracted, e.g., $|\mathcal{V}|$, $|\mathcal{E}|$ and Δ of $\mathcal{G}$. In addition, if the graph is labelled, the vertex frequency of each label is computed (see usage for FSM in §7.2). After $\mathcal{G}$ is loaded into memory, some preprocessing is performed on $\mathcal{G}$. First, the neighbor list of each vertex is sorted by ascending order of vertex IDs, so that we can apply early exit when we search the list with an upper bound (i.e., symmetry breaking). Second, if a pattern of `clique` is detected, G^{2}Miner enables a typical optimization called *orien-*

Optimizations		Effect					Used in Pangolin?	Used in hand written apps?	Conditions to apply
		mitigate divergence	load balance	mem. saving	algorithm pruning	extra GPU efficiency			
Category-(1): Known	A: Data graph preprocessing (edge orientation) §4.2			✓	✓		✓	✓	cliques
Category-(2): Known, but not enabled in prior GPM systems	B: Data graph partitioning §7.2 (1)			✓			×	TC only	hub patterns, graph size, GPU memory size
	C: Two-level parallelism §5.1	✓	✓				×	TC only	always enabled on GPU
	D: Counting-only pruning §5.4 (1)				✓		×	CPU only	automatic pattern decomposition [82]
	E: Local graph search §5.4 (2)				✓		×	CL only	hub patterns & Δ <1024
	F: Flexible data format §6.2	✓					×	CC only	
	G: Multi-gpu scheduling §7.1		✓				×	MC only	always used on multi-GPU
Category-(3): Novel for GPM	H: SIMD-aware primitives §6.1					✓	×	×	hardware support for warp level primitives
	I: Multi-pattern fission §5.3					✓	×	×	explicit multi-pattern & kernel occupancy by NVCC
	J: Edgelist reduction §7.2 (2)			✓			×	×	if $v_0 > v_1$ in symmetry order
	K: Adaptive buffering §7.2 (3)			✓			×	×	buffer W usage in matching order & GPU memory size
	M: Hybrid order on GPU §5.2			✓			×	×	implicit, intermediate data unbounded, user-specified
	N: memory reduction using label frequency §7.2 (4)			✓			×	×	implicit, vertex label frequency, user-specified

Table 2: Optimizations in G2Miner. Among them, optimizations A, B, D, E, F, I, J, K, M, N are pattern-aware; optimizations B, C, G, H, I, K, M are architecture-aware; and optimizations B, E, F, K, N are input-aware. Pattern-aware optimizations are applied based on the pattern analysis, while input-aware and architecture-aware optimizations are enabled according to the input and architecture information, respectively. TC: triangle counting. CL/CC: clique listing/counting. MC: motif counting.

tation [26] . It gives every edge a direction in the undirected data graph $\mathcal{G}$, which in turn converts $\mathcal{G}$ into a directed graph. This halves the edge count in $\mathcal{G}$, significantly reduces Δ, and completely eliminates on-the-fly checking. Third, our preprocessor also supports sorting (e.g., by degree) and renaming the vertices in $\mathcal{G}$ to improve load balance [53, 73]. Note that all these preprocessing operations need to be done only once.

4.3 Overview of Optimizations

Table 2 lists all the optimizations enabled in G^{2}Miner. We classify them into three categories. Optimizations in Category-(1) are those exist in prior GPM systems. Optimizations in Category-(2) do not exist in prior GPM systems (e.g., Pangolin) but have been used in some hand-written GPM applications. For example, optimization `D: data graph partitioning` has only been used for triangle counting, while in G^{2}Miner we generalize it for all the clique patterns. These optimizations are missing in prior GPM systems because prior systems are oblivious to the required pattern, input or architecture information. Optimizations in Category-(3) are novel as they have never been used for GPM, though some of them are known for GPU computing in general.

As shown in column 3 to 7 of Table 2, these optimizations have different kinds of effect on GPM applications: (1) mitigating thread divergence; (2) improving load balancing; (3) reducing memory consumption; (4) pruning search space; and (5) improving efficiency based on GPU hardware features.

The last column of Table 2 shows the conditions for each optimization to be applied. All the optimizations in Table 2 are automated in G^{2}Miner based on detecting the conditions, except for M and N (the last two rows). M and N are particularly used for implicit-pattern problems like FSM, for which the system cannot infer the conditions automatically. Thus, M and N are user-activated by specifying a flag.

Next, we describe these optimizations in detail, in the three major components of G^{2}Miner: the code generator (§5), the device function library (§6) and the runtime scheduler (§7).

5 Pattern-specific GPU Code Generation

G^{2}Miner includes a *pattern-aware* code generator that automatically generate CUDA code specific to the pattern. Prior work [73, 74] has explored how to generate pattern-specific CPU code based on the matching order and symmetry order, but code generation is more challenging for GPU.

Generating pattern-specific CPU code is relatively straightforward. For example, to generate Algorithm 1 for `diamond`, the matching order in Fig. 5 (a) is used to generate the 4 nested `for` loops, and the symmetry order is then used to insert breaks at Line 3 and 7. Whenever a set operation is needed, a function call to the set operation primitive (imple-

mented in a library) is inserted (Line 4). Since v_3 and v_4 are both from $\mathcal{N}(v_1) \cap \mathcal{N}(v_2)$, a buffer W is created for data reuse. Finally, *task parallelism* is used to parallelize the program, i.e., each thread processes one task at a time (Line 1).

However, generating efficient GPU code is more challenging, because (1) DFS-based GPM suffers from the thread divergence and load imbalance issues (§5.1); (2) hybrid search orders are needed in some cases (§5.2); and (3) extra support is needed for multi-pattern problems (§5.3) and advanced pruning schemes (§5.4).

5.1 Parallel Strategies for DFS on GPU

To maximize GPU efficiency for the DFS algorithm, we employ a *two-level parallelism* strategy in G^{2}Miner to exploit both inter-warp task parallelism and intra-warp *data parallelism*. This is motivated by our key observation that in GPM algorithms *most of execution time is spent on set operations*. For example, when we executed Peregrine on a multicore CPU, set operations for each benchmark took 75% to 92% of the total execution time. This motivated us to parallelize set operations by exploiting the data parallelism within each warp. It alleviates divergence and also provides more parallelism to fully utilize GPUs. To reduce load imbalance and further increase parallelism, we use edge parallelism for GPU instead of the vertex parallelism used for CPU.

(1) Reduce divergence with warp-centric parallelism. We could map each task to a thread, a warp or a CTA in a GPU. As DFS has much more coarse-grained tasks than BFS, mapping a task to a thread would be highly divergent and unbalanced for GPUs. However, if we map a task to a CTA, all (e.g., 256) threads in the CTA will be used to process the same set operations. If the two input neighbor-lists of a set operation are small, many threads in the CTA will be idle, leading to low utilization. Moreover, all threads in the CTA will do the same DFS walk, which is a lot of redundant computation.

In G^{2}Miner we use *warp-centric data parallelism*. Each task is assigned to a warp. All threads in a warp synchronously perform the same DFS walk of the task. During the DFS walk, whenever a set operation is encountered, all threads in the warp work cooperatively to compute the set operation in parallel. It has several benefits. First it achieves higher throughput than CPU since set operations are parallelized. Second, it alleviates thread divergence within each warp as all threads in a warp are progressing synchronously. Third, it causes less redundancy than using CTA. Our evaluation shows it is on average 2× faster than CTA-centric parallelism.

(2) Reduce task granularity for load balance. GPM systems on CPU use vertex parallelism [25, 53, 73, 74], i.e., each task is a DFS walk rootedin a vertex, as shown in Fig. 6 (a). This can already provide enough parallelism for CPU, needs no auxiliary data, and potentially enjoys data reuse within the sub-tree. But the coarse-grain tasks lead to load imbalance which can not be well tolerated by GPUs. To reduce task granularity, we use edge-parallelism, i.e., each task explores the subtree rooted by an edge. As shown in Fig. 6 (b), apparently more work is required to search the subtree below a vertex on average compared to searching the subtree below an edge. In addition to better load balance, edge parallelism can provide more parallelism ($|\mathcal{E}| > |\mathcal{V}|$) for GPU than vertex parallelism.

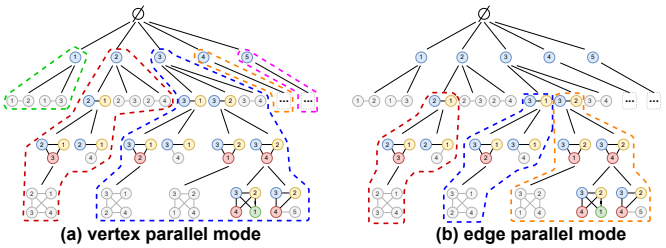


Figure 6: (a) vertex-parallel vs. (b) edge-parallel execution. Each dashed circle is a parallel task. A task is mapped to one thread on CPU, but in G^{2}Miner it is mapped to one warp on GPU.

- By default, our code generator generates edge-parallel kernels. Our evaluation shows they are mostly (1.5× on average) faster than vertex parallel ones. But some GPM algorithms must use vertex parallelism. For example, the 3-MC algorithm in [25] can only be done in vertex parallelism. G^{2}Miner supports both vertex and edge parallelism. The user can set a compiler flag to use vertex parallelism, in which case Ω is not generated to save memory.

Discussion. *Two-level parallelism* has been only used for triangle counting [50], and it is challenging to extend it for all GPM problems. First, triangle counting does subgraph extension only once, which needs no DFS traversal. Thus, G^{2}Miner is the first to support DFS for GPM on GPU. Second, naive GPU implementations for complex patterns can easily run out of memory for intermediate data. This is not a concern for triangle counting. Third, during the DFS traversal, it requires extension to support high-performance generic set operations and multi-pattern, which triangle counting does not require. In the following, we show that these challenges can be resolved by applying optimizations H, I, J, K, M, N in Table 2.

5.2 Support for Hybrid Search Orders

With the two-level parallelism in §5.1, for many GPM problems, DFS is faster than BFS in G^{2}Miner. However, this is not the case for problems like FSM. FSM computes the `domain support` and thus requires aggregating all the subgraphs for each pattern to compute its support. In the DFS-based FSM algorithm [99, 114], each task is a single-edge pattern (instead of subgraph) and the entire subtree of that pattern. This is *pattern-parallel*, instead of vertex-parallel or edge-parallel. Since the number of patterns is much smaller than the number of vertices or edges, the parallelism in FSM is not sufficient for GPU. Moreover, the task granularity in pattern-parallelism is much larger than that in vertex- or edge-parallelism, making

Algorithm 3 Pseudo code for counting `diamond`

```
1: for each vertex v1 ∈ V in parallel do                ▷ match v1 to u1
2:     for each vertex v2 ∈ N(v1) do                    ▷ match v2 to u2
3:         if v2 ≥ v1 then break;                       ▷ symmetry breaking
4:         n = |N(v1) ∩ N(v2)|;                         ▷ # triangles incident to (v1, v2)
5:         count += n*(n-1)/2                           ▷ choose 2 from n to form a diamond
```

the problem even more unbalanced.

In G^{2}Miner we use a hybrid of BFS and DFS, or *bounded BFS* search for problems that use `domain support` (e.g., FSM). At the single-edge level (i.e., level-2), we start with BFS search to aggregate edges by their patterns in parallel, which provides abundant parallelism. As the search goes deeper, the number of subgraphs increases exponentially. To fit the intermediate data in memory, we divide the subgraphs into blocks. Each block has a size that can resides in GPU memory, but also contains enough amount of subgraphs that can fully utilize the GPU. Once the current block is processed, it moves to the next block. Using this bounded BFS search, G^{2}Miner can support larger graphs than Pangolin.

5.3 Support for Multi-pattern Problems

Multiple patterns may have a common sub-pattern, which can be shared if they are searched in the same CUDA kernel. On the other hand, mining multiple patterns simultaneously would need a significant amount of intermediate resources, e.g., registers, which results in low hardware utilization (occupancy) on GPU.

Instead of generating a single gigantic kernel for all patterns, we employ *kernel fission* to reduce register pressure. Given multiple patterns, we leverage pattern analysis to find which patterns share the same sub-pattern, so that they should be merged into the same kernel to enjoy sharing. For those patterns do not share the same sub-patterns, we generate different kernels for them, so that each kernel is lightweight enough to avoid high register pressure. For example, in 4-motifs (Fig. 3), `tailed-triangle`, `diamond` and `4-clique` share the same sub-pattern `triangle`. So we generate a single CUDA kernel for the three patterns, in which they share the same workflow that enumerates triangles. However, for the other patterns, since there is no sharing opportunity, we generate one kernel for each. These separated kernels use fewer registers than a combined kernel, so that each SM in GPU can accommodate more co-running warps to maximize utilization. This improves performance by 15% for mining 4-motifs.

5.4 Support for Advanced Pruning Schemes

(1) Counting-only Pruning. If the user is interested in *counting* instead of *listing* subgraphs, there may exist an advanced pruning opportunity to further reduce the search space. For example, to count edge-induced `diamond` (Algorithm 3), because a `diamond` consists of two triangles, we first compute the triangle count n for each edge (v_1, v_2) using set intersection (Line 4), and then use the formula $\binom{n}{2} = n \times (n-1)/2$ to get the `diamond` count (Line 5). Note that this pruning opportunity is pattern specific and is not always available. For example, there is no such opportunity for `4-cycle`. Our pattern analyzer detects the opportunities by using automatic pattern decomposition [21,82], and based on the detection, our code generator can accordingly generate the CUDA kernel.

(2) Local Graph Search (LGS). This is a pruning scheme used for `hub-patterns`. A hub-pattern contains at least one `hub` vertex that is connected to all other vertices. For example, any vertex in a `clique` is a hub vertex. The key idea of LGS is, instead of searching a massive data graph $\mathcal{G}$, we can construct a small local graph for each vertex in $\mathcal{G}$ and search in the local graphs. For a hub pattern with a hub vertex u_1, we match the first data vertex v_1 to u_1, and the entire sub-tree rooted by v_1 is confined within v_1's 1-hop neighborhood. Fig. 7 shows an example of constructing a local graph. Search in the local graph is faster because the vertex degrees in the local graph are smaller than those in the global data graph. When the pattern analyzer detects a hub-pattern, the code generator inserts a call to construct local graphs, and generates code to search in the local graphs, instead of the original data graph.

- Previously, LGS has only been used for clique patterns [30], while G^{2}Miner generalizes and automates it for all hub patterns. Moreover, unlike CPUs, naive implementation on GPUs is not beneficial. We combine LGS with the `bitmap` format (see §6.2) to achieve significant speedups.
- *Input Awareness*. LGS is not always beneficial [25]. The key indicator is the maximum degree Δ of the data graph. For example, if Δ is too large, it is not beneficial due to high overhead of local graph construction. Therefore, we generate CUDA kernels for both cases: LGS enabled and disabled. The runtime system checks if Δ is above a threshold and decides accordingly which kernel to use. LGS brings us 1.2 $\sim$ 3.7$\times$ speedup on GPU for various data graphs.

6 Device Primitives for Set Operations

As G^{2}Miner assigns each task to a warp, whenever there is a set operation, all the thread in a warp work cooperatively to compute it. For example, in Algorithm 1, there is a set intersection at Line 4. In G^{2}Miner, set operations are done by invoking the corresponding device functions predefined in the GPU primitive library. We leverage GPU hardware SIMD support to implement efficient set operations (§6.1) and flexibly support various data formats for vertex sets (§6.2).

6.1 SIMD-aware Primitives

Given two sets A and B, we need two major set operations in GPM: (1) set intersection: $C = A \cap B$; (2) set difference: $C = A - B$, where C is the output set. Besides, another operation

set bounding is also often needed: given a set A and an upper bound y, set bounding computes $\{x|x < y \& x \in A\}$. We discuss set intersection in detail, and the other operations are similar.

In Algorithm 1 Line 4, the result of set intersection is stored in a buffer W for reuse. Buffering is widely used in GPM algorithms to avoid repetitive computation [74]. To support buffering in G^{2}Miner, each warp is allocated a private buffer in the GPU memory. In the primitive functions, threads in a warp write outputs to the buffer in parallel. To do this efficiently, we use CUDA warp-level primitives [69] which are supported by the GPU hardware (special instructions). For each vertex v in set A, we use a boolean flag to indicate whether it exists in set B. Using the flag, we compute a mask using `__ballot_sync` primitive. The mask is then used to compute the index and the total size of the buffer using `__popc` primitive.

Implementation details. Previous work has explored set intersection for SIMD [10, 15, 45, 51, 118] or GPU [8, 38, 40, 41, 49, 50, 79, 80, 112, 113]. We classify their algorithms into 3 categories: *Merge-path* [40, 41], *Binary-search* [38, 50] and *Hash-indexing* [80]. We have extensively evaluated these methods on GPU, and we find that binary search works the best since it is less divergent. In our library, we implement a high-performance binary search [50]: to exploit temporal locality, we leverage the scratchpad in GPU to pre-load the first five layers of the binary search tree, which further mitigates memory divergence. We extend this method to also support set difference, set bounding, and local graph construction.

6.2 Flexible Data Representation

Vertex set is a key data structure in GPM, which is used for the neighbor list in $\mathcal{G}$ and the buffer W in Algorithm 1. Its representation on GPU has a major impact on performance. For the set operations particularly, using a dense representation makes set operations easy to compute, but it requires more storage space. If using a sparse representation, it saves space but complicates the computation of set operations.

We support two types of formats for vertex set on GPU: `sorted-list` (sparse) and `bitmap` (dense). `sorted-list` is a list (i.e. array) of vertices sorted in ascending order. `bitmap` is a sequence of bits (length=$|\mathcal{V}|$), each of which indicates the connectivity to a vertex in $\mathcal{V}$. Set operations on `bitmap` are very simple and efficient, but `bitmap` consumes more space when $\mathcal{V}$ is large. Thus, by default we use `sorted-list`, and we only enable `bitmap` for hub-patterns since the `bitmap` size can be reduced significantly (Δ instead of $|\mathcal{V}|$).

7 Runtime Scheduling and Management

Our runtime system is aware of the pattern, input data graph and GPU architecture to balance workload among multiple GPUs (§7.1) and make full use of the GPU memory (§7.2).

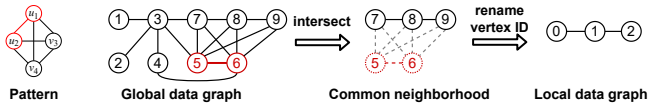


Figure 7: Local graph constructed for v_1=5 and v_2=6 which are matched to hub vertices u_1 and u_2 in the pattern respectively. We first compute set intersection of vertex 5 and 6, to get their common neighbors (vertex 7, 8, 9). The common neighbors are renamed to form a local graph. Renaming can reduce `bitmap` storage.

7.1 Task Scheduling for Multi-GPU

Given n as the number of GPUs[1] available in the system and a data graph $\mathcal{G}$ with an edgelist $\Omega = \{e_1, e_2, ..., e_m\}$ where $m = |\mathcal{E}|$ (in the case of symmetry breaking at level 2, $m = |\mathcal{E}|/2$), the task scheduler aims to divide GPM computation onto the n GPUs, by dividing Ω into n segments, each of which has the same amount of work, such that the execution time of the last completed GPU is minimized.

BFS-based GPM systems, e.g., Arabesque, RStream, and Pangolin, balance workload by reassigning tasks at every level. But this does not work for the DFS algorithm because DFS does not work in the level-by-level way as BFS. Existing DFS-based GPM systems target only CPUs, and thus can use sophisticated work stealing techniques [33]. But this will incur non-trivial runtime overhead on multi-GPU ($\sim$20%) [23, 50].

Policy 1: Even-split Scheduling. Ω is to evenly split into n consecutive ranges, each of which contains m/n tasks. This is used in existing triangle counting solvers on multi-GPU [80]. This policy is simple and has no scheduling overhead, but it results in severe load imbalance for skewed graphs. Fig. 8 shows the time spent on each GPU to finish its work under the even-split scheme. Due to the skewness of the workload assigned to each GPU, under the 2-GPU setting we observe that GPU_0 takes much more time to finish its work than GPU_1. The same time variance is observed for the 3-GPU and 4-GPU setting. Worse still, in the 4-GPU setting, since most of the heavy tasks are assigned to GPU_1, it makes the 4-GPU setting even slower than the 3-GPU setting. This means the even-split scheme does not scale beyond 3-GPU for this benchmark. The reason of poor scalability is two-folds: (1) the granularity of splitting workload is too coarse-grain; (2) it is unaware to the skewness of task workload by assuming every task has the same amount of work.

Policy 2: Round-robin Scheduling. Each GPU has a task queue, denoted as Q_i for the i-th GPU, $i \in [0, n)$. The tasks in Ω are assigned to each queue in a round-robin fashion, i.e., e_j is assigned to Q_i, where $i = j \bmod n$, $j \in [0, m)$. This is a fine-grained scheduling policy that has been used in existing motif counting solvers on multi-GPU [89]. The policy comes with some overhead, i.e., copying tasks into task queues. This copy is needed only once for a specific data graph and n, i.e., once

[1]We assume that every GPU has the same compute power for simplicity, otherwise it is not difficult to scale the workload by a factor accordingly.

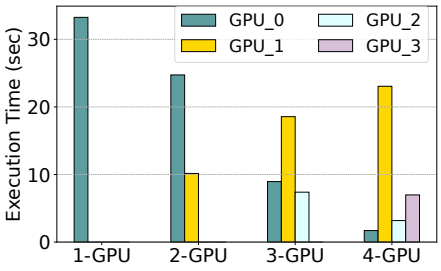


Figure 8: Running time of each GPU using even-split: 3-MC on Tw2.

done, the queues can be reused for mining different patterns.

Policy 3: Chunked Round-robin Scheduling. Ω is first split into lots of small chunks, and then we assign chunks to the task queues in a round-robin way. This is a generalization of the previous two policies. When the chunk size $c = m/n$, it becomes the same as policy 1. When $c = 1$, it becomes the same as policy 2. Thus if c is too small the data copying overhead will be high, but if c is too large, we see load imbalance as in policy 1. We use $c = \alpha \times y$, where y is the total number of warps and α is a constant (set to 2 empirically). Our chunking is also pattern aware, as described in §7.2.

Implementation details. To further reduce data copy overhead, we parallelize it as the location to copy to is fixed for each queue if the chunk size is fixed. Note that this overhead is constant to the pattern size k, which is trivial (< 1%) when $k > 3$, since the GPM computation is exponential to k. For small pattern like triangle, we overlap the scheduling overhead with the GPU computation, by first assigning a few chunks to each GPU and launch the kernel. During the GPU computation, we continue sending the remaining chunks from the CPU to feed the GPUs. Orthogonal work on ordering tasks in Ω [89] or grouping tasks by community may help further improve load balance and locality.

7.2 GPU Memory Management

GPU memory is a scarce resource. In GPM algorithms, the major memory usage involves the data graph $\mathcal{G}$, the edgelist Ω and the buffers (e.g., W in Algorithm 1)). For FSM, the subgraph list of each pattern requires additional space.

(1) Preprocessing the data graph. We have discussed *orientation* in §4.2. In the multi-GPU setting, for any hub-pattern, since the search is confined in the root vertex v_1's neighborhood, we partition $\mathcal{V}$ into n subsets (n is the number of GPUs). For each i-th subset we generate its vertex induced subgraph of $\mathcal{G}$, and copy it to the i-th GPU. This partitioning reduces memory usage and guarantees that there is no communication needed between GPUs. This technique has been used in [47] only for triangle counting. We generalize it for all hub-pattern problems. The scheduling policy is then adjusted by chunking vertices and assigning incident edges in Ω to the corresponding GPUs. For non-hub patterns, we do not partition $\mathcal{G}$ if it can fit in the single-GPU memory. This is because GPM algorithms access multi-hop neighbors, which leads to non-trivial communication overhead [99], especially for small-diameter graphs. When $\mathcal{G}$ is too large to fit in memory, we leverage community-aware partition [56] to minimize communication.

(2) Reducing the size of edgelist. For Ω, we apply an important optimization by considering symmetry at the edge level (level-2). Since $\mathcal{G}$ is an undirected graph, for each undirected edge in $\mathcal{G}$, the edgelist contains two instances, each for one of the two directions of the edge. However, when there is a partial order between v_1 and v_2 for symmetry breaking, we generate the edgelist that contains only one instance. More specifically, if $v_1 > v_2$ is included in the symmetry order (e.g., in Fig. 5 (b)), the edgelist includes only the edges whose source vertex id is larger than its destination vertex id. In this way, we can reduce half of the edges before execution. It not only saves memory but also reduces checking on-the-fly. Note that there is a similar optimization [96] to split the neighbor list of each vertex v into two sets, with one holding all neighbors whose IDs are larger than v, and the other holding the rest which have smaller IDs than v. This reduces on-the-fly checking, but it is not used to reduce memory usage.

(3) Adaptive buffering. In G^{2}Miner's warp-centric DFS walk, each warp is allocated with X buffers. The value of X is pattern specific and the pattern analyzer can decide it when generating the search plan. For a pattern of size k, $X \leq k-3$ because the first two levels and the last level do not need buffers. So the worst case memory consumption for buffering is $O(\Delta \times (k-3))$. This is linear to k for a given specific data graph. In comparison, the intermediate data generated in Pangolin is exponential to k, which can be easily over the GPU memory capacity (see in §8.1). Although Δ is much smaller than $\mathcal{E}$ (see Table 3), given the large number of warps in GPU, the memory space for buffers can still be very large. Therefore, the runtime limits the total number of warps to save memory usage, so that all tasks assigned to the same warp share the buffer usage. In this way, given different data graphs, we can adaptively tune the number of warps to make full use of memory and maximize parallelism. More specifically, we subtract the size of $\mathcal{G}$ and Ω from the total GPU memory size, to get the remaining memory size, denoted as Y. Then we can get the maximum number of warps $Y/(X \times \Delta)$. Finally we launch $\texttt{min}(Y/(X \times \Delta), |\Omega|)$ warps.

(4) Reducing memory allocation using label frequency. This optimization is particularly useful for problems that find *frequent patterns*, such as FSM. The graph loader in G^{2}Miner computes the vertex frequency for each label. This information can be leveraged to find *frequent labels*, i.e., labels with vertex frequency above the user-defined support threshold σ_{min}. Since infrequent labels can not be part of frequent patterns, the total number of possible frequent patterns N can be significantly reduced, if there are many infrequent labels. Note that in FSM we allocate a *subgraph list* for each possible pattern to store subgraphs for aggregation, and the memory consumption of these subgraph lists is proportional to N. With

Graph	Source	|V|	|E|	Label	Max deg. Δ
Mi	Mico [35]	0.1M	2M	29	1,359
Pa	Patents [44]	3M	28M	37	789
Yo	Youtube [28]	7M	114M	28	4,017
Lj	LiveJournal [66]	4.8M	43M	0	20,333
Or	Orkut [66]	3.1M	117M	0	33,313
Tw2	Twitter20 [63]	21M	530M	0	698,112
Tw4	Twitter40 [64]	42M	2,405M	0	2,997,487
Fr	Friendster [115]	66M	3,612M	0	5,214
Uk	Uk2007 [16]	106M	6,603M	0	975,419

Table 3: Data graphs (symmetric, no loops or duplicate edges). Maximum degrees are smaller when orientation is applied for cliques.

Data Graph	Lj	Or	Tw2	Tw4	Fr	Uk
G^{2}Miner (GPU)	0.03	0.14	1.6	5.1	3.2	7.5
Pangolin (GPU)	0.06	0.25	3.0	OoM	5.2	OoM
PBE (GPU)	0.27	1.12	13.4	53.5	23.0	55.3
Peregrine (CPU)	1.63	7.25	112.1	8492.4	100.3	3640.9
GraphZero (CPU)	0.61	2.22	24.4	1399.3	49.0	1041.3

Table 4: TC running time (sec). OoM: out of memory.

this awareness of the input (i.e., label frequency), we can drastically reduce this memory consumption in many cases.

8 Evaluation

We compare G^{2}Miner[2] with state-of-the-art systems: (1) GPM system on GPU, Pangolin [26], (2) subgraph matching solver on GPU, PBE [42, 43], (3) CPU-based GPM system Peregrine [53] and (4) CPU-based subgraph matching system GraphZero [73, 74]. Note that Pangolin also provides a CPU implementation, but it is slower than GraphZero.

Table 3 lists the data graphs. The first 3 graphs (`Mi`, `Pa`, `Yo`) are vertex-labeled graphs which are used for FSM. We use all the GPM applications listed in §2.1 for evaluation, i.e., TC, k-CL, SL, k-MC. For SL, we use two patterns `4-cycle` and `diamond`. Note that GraphZero does not support FSM, Pangolin does not support SL, and PBE does not support k-MC and FSM. For FSM, we include DistGraph [99] in Table 8 as the state-of-the-art hand-written FSM solver.

CPU-based systems and solvers are evaluated on a 4 socket machine with Intel Xeon Gold 5120 2.2GHz CPUs (56 cores in total) and 190GB RAM, while GPU-based solutions are evaluated on NVIDIA V100 GPUs (each with 32GB device memory). We exclude preprocessing (e.g., DAG construction in Pangolin and vertex reordering in Peregrine) time in all systems. We use a time-out of 30 hours for CPU and 8 hours for GPU, and report all results as an average of three runs. We show single-GPU performance in §8.1 and compare with CPU solutions in §8.2. Multi-GPU performance of G^{2}Miner is shown in §8.3. Impact of optimizations is analyzed in §8.4.

8.1 Single-GPU Performance

We compare with Pangolin and PBE on a V100 GPU. Table 4 lists the GPU running time for triangle counting (TC). We

[2]G^{2}Miner source code: https://github.com/chenxuhao/GraphMiner

Pattern	4-CL					5-CL		
Data Graph	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Fr
G^{2}Miner (G)	0.32	0.54	113.3	362.9	7.3	3.2	1.7	13.1
Pangolin (G)	1.48	4.04	OoM	OoM	OoM	OoM	OoM	OoM
PBE (G)	3.90	11.11	3640.1	TO	117.8	246.4	99.2	399.8
Peregrine (C)	15.90	73.70	39921.0	TO	397.3	520.8	782.1	957.6
GraphZero (C)	3.48	12.96	2152.2	20591.1	177.7	60.0	48.3	243.3

Table 5: k-CL running time (sec). TO: timed out.

Pattern	Diamond					4-cycle		
Data Graph	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Fr
G^{2}Miner (G)	0.29	0.75	26.8	183.1	12.8	2.7	33.7	1291.2
PBE (G)	0.48	1.71	26.3	102.0	39.9	17.3	177.8	5211.3
Peregrine (C)	5.38	10.24	553.6	20898.4	178.1	144.4	1867.2	32276.8
GraphZero (C)	1.73	7.27	165.1	7938.6	136.4	34.0	345.5	9251.5

Table 6: SL running time (sec). 'G': GPU; 'C': CPU.

observe that Pangolin runs out of memory for `Tw4`[3] and `Uk`, while G^{2}Miner can run with all the data graphs. We also observe that G^{2}Miner is constantly faster than Pangolin, due to optimized set operations in our library. On average, G^{2}Miner is 1.8× faster than Pangolin on V100 GPU.

The speedups are more significant for k-CL and k-MC. As shown in Table 5, G^{2}Miner outperforms Pangolin by 4.6× and 7.6× for 4-clique listing on `Lj` and `Or` respectively. The speedups mainly come from data reuse enabled in DFS (i.e., buffering W in Algorithm 1) and optimized set operations[4]. Meanwhile, for all the rest of graphs and the larger pattern 5-clique, Pangolin runs out of memory. Similar trend is found in Table 7, where we observe an average of 21.3× speedup over Pangolin on 3-MC, and Pangolin also runs out of memory for most of the cases. G^{2}Miner managed to run all cases, which demonstrates that its DFS order and optimization J and K in Table 2 can effectively reduce memory consumption.

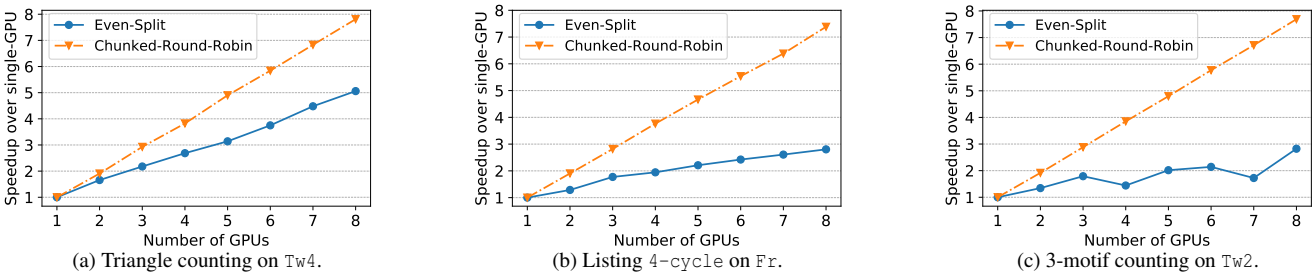
For FSM in Table 8, G^{2}Miner is competitive with Pangolin for the small graphs, since we use bounded BFS (optimization M in Table 2) that provides enough parallelism. For the largest graph `Yo`, Pangolin runs out of memory again, while G^{2}Miner succeeds to run it, thanks to both optimization M and N in Table 2 which help reduce memory consumption.

Overall, G^{2}Miner achieves an average speedup of **5.4**× over Pangolin, and the speedup is more significant for larger patterns. Moreover, G^{2}Miner can run much larger graphs.

We also compare with PBE [42, 43] on the V100 GPU. PBE partitions the data graph when it gets large, which allows it run all the single-pattern workloads. However, its performance is even worse (3.8× slower) than Pangolin, due to the cross-partition communication overhead and lack of data graph orientation. Particularly, for subgraph listing, as `diamond` contains a sub-pattern `triangle` but `4-cycle` does not, searching `diamond` generates much less intermediate data than searching `4-cycle`. Thus in Table 6 we observe that PBE's `4-cycle` performance is much worse than G^{2}Miner as

[3]Since data graphs are oriented in TC, `Fr` takes less memory than `Tw4`

[4]It can not be directly used for Pangolin, as Pangolin maps *connectivity checks* [26] to threads, but G^{2}Miner maps set operations to warps.



(a) Triangle counting on `Tw4`. (b) Listing `4-cycle` on `Fr`. (c) 3-motif counting on `Tw2`.

Figure 9: G^{2}Miner multi-GPU scalability using two task scheduling policies: even-split vs. chunked-split.

Pattern	3-Motif					4-Motif		
Data Graph	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Fr
G^{2}Miner (G)	0.17	0.97	33.3	1703.6	22.0	138.1	2068.4	15475.4
Pangolin (G)	2.05	22.62	1165.5	OoM	OoM	OoM	OoM	OoM
Peregrine (C)	9.36	19.46	418.7	27954.9	367.9	1435.4	20219.1	TO
GraphZero (C)	1.50	7.74	276.5	7439.4	169.6	3039.6	16394.6	TO

Table 7: *k*-MC running time (s). OoM: out of mem.; TO: timed out.

Data Graph	Mico				Patent				Youtube			
σ	300	500	1000	5000	300	500	1000	5000	300	500	1000	5000
G2Miner (G)	0.6	0.4	0.3	0.1	2.6	2.6	2.6	1.7	7.2	6.0	6.0	8.7
Pangolin (G)	0.6	0.5	0.3	0.2	2.7	2.7	2.7	1.7	OoM	OoM	OoM	OoM
Peregrine (C)	4.4	4.4	4.2	4.3	94.2	103.8	118.4	94.3	59.3	52.8	69.9	60.8
DistGraph (C)	56.1	61.0	57.6	57.0	13.2	13.1	13.0	14.1	OoM	OoM	OoM	OoM

Table 8: 3-FSM running time (sec). OoM: out of memory.

it has to do partitioning and suffers from the overhead. Overall, G^{2}Miner achieves a **7.2**× speedup over PBE on average.

8.2 Mining on GPU vs. on CPU

To evaluate how much speedup we can get from GPU over CPU, we compare G^{2}Miner (on V100 GPU) with GraphZero (on 56-core CPU). Note that for each specific GPM application, G^{2}Miner and GraphZero use exactly the same matching order and symmetry order, making it a fair comparison to show the benefit from the difference of hardware architectures. As listed in Table 4, G^{2}Miner is significantly faster than GraphZero on TC, with an average speedup of 38.0×. The same trend is observed for *k*-CL in Table 5, where G^{2}Miner outperforms GraphZero by 18.2×. This tremendous performance improvement is due to three parts: (1) the orientation optimization, (2) higher throughput (i.e. more parallelism) on GPU, and (3) our high-performance set operations on GPU.

For SL, orientation can not be applied. Thus it can be used to evaluate the benefit of the other two parts. As shown in Table 6, G^{2}Miner still achieves overwhelmingly better performance than GraphZero, with an average speedup of 10.5×. The speedup would be marginal if we use the BFS strategy in Pangolin and PBE or implement our DFS scheme naively.

While TC, *k*-CL and SL uses only set intersection, *k*-MC includes both set intersection and set difference. As G^{2}Miner optimizes both operations, we also observe dramatic performance boost for *k*-MC. In Table 7, it constantly outperforms GraphZero for all benchmarks. On average G^{2}Miner is 8.5× faster than GraphZero.

Overall, G^{2}Miner on GPU achieves **15.2**× speedup over GraphZero on CPU, which demonstrates the significant benefit of using GPU to accelerate GPM applications.

As GraphZero does not support FSM, we also compared to Peregrine. G^{2}Miner on GPU is **48.3**× faster than Peregrine on CPU. Note that Peregrine does not mine multiple patterns simultaneously for multi-pattern problems. Instead, for *k*-MC and FSM, it enumerates every pattern one by one, making it impossible to reuse data across similar patterns. Thus it is mostly even slower than GraphZero.

8.3 Multi-GPU Scalability

We evaluate multi-GPU performance by varying the number of GPUs from 1 to 8 in a single machine. Since PBE and Pangolin do not support multi-GPU, we only evaluate G^{2}Miner in this section. We compare two task scheduling policies in Fig. 9. As illustrated, the chunked round-robin scheme constantly works much better than the even-split scheme. More importantly, the chunked scheme scales linearly for all cases, while the even-split scheme fails to scale beyond 3-GPU for 3-MC on `Tw2`. The poor scalability of even-split is dues to the load imbalance. As shown in Fig. 10, in the 4-GPU setting, the execution time of each GPU varies dramatically for the even-split setting. In contrast, for the chunked scheme, each GPU finishes its work roughly at the same time.

8.4 Impact of Optimizations

Different optimizations in Table 2 contribute differently to the performance improvement. First, architecture-aware optimizations are crucial for all workloads on GPU. G^{2}Miner is 5.4× faster then Pangolin, where *two-level parallelism* (C in Table 2) and *SIMD-aware primitives* (H in Table 2) contribute 3.1× and 1.7× respectively. Second, for a pattern-aware optimization, it is beneficial only for the target pattern(s), and the speedups vary a lot depending on how much the search space is pruned. For example, *local-graph search* (E+F in Table 2) brings 1.2× ∼ 3.7× speedup for `hub-patterns` (2.1× on av-

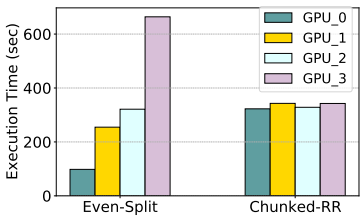


Figure 10: Running time of each GPU in the 4-GPU setting: `4-cycle` on `Fr`.

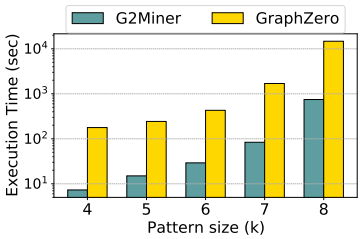


Figure 11: Running time of k-clique listing over `Fr`, $k \in [4,8]$.

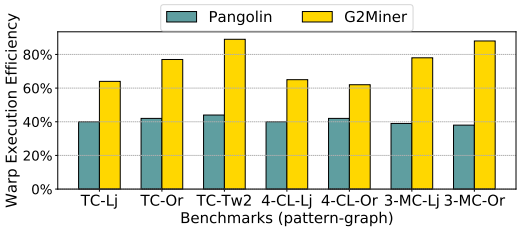


Figure 12: Warp execution efficiency.

Pattern	Diamond					3-Motif					4-Motif		
Time (sec)	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Tw2	Tw4	Fr	Lj	Or	Fr
G^{2}Miner (GPU)	0.09	0.47	9.9	66.9	10.4	0.06	0.27	6.8	21.4	5.2	2.6	34.2	1307.2
Peregrine (CPU)	2.20	8.66	245.8	16312.6	158.8	2.51	4.90	116.0	8447.4	165.3	163.6	1701.4	TO

Table 9: Running time of G^{2}Miner vs. Peregrine, both with counting-only pruning enabled. TO: timed out.

erage), while *counting-only pruning* (D in Table 2) achieves 1.2× (`diamond`, `Fr`) to 79.7× (`3-motif`, `Tw40`), with 6.2× on average. Other optimizations in Table 2 are for memory saving, which is crucial for enabling larger datasets.

Large Pattern and Large Graph. A major advantage of G^{2}Miner over Pangolin is that G^{2}Miner can support much larger graphs and patterns. Fig. 11 shows that G^{2}Miner can run up to 8-clique listing on a billion-edge graph `Fr`. In contrast, Pangolin can not even run 4-clique due to out-of-memory, as shown in Table 5. Fig. 11 also shows that, from 4-clique to 8-clique, G^{2}Miner on GPU consistently achieves an order of magnitude speedup over GraphZero on the CPU, although the GPU has much less memory than the CPU. This trend implies that GPUs can be not only capable but also highly efficient for processing large graphs and patterns, thanks to G^{2}Miner's memory management and optimizations for the GPU architecture.

GPU Efficiency. To evaluate GPU utilization, we measure *warp execution efficiency*, which is the average percentage of active threads in each executed warp. As shown in Fig. 12, the warp execution efficiency in Pangolin is around 40%. This is relatively low since more than half of the compute horse power is wasted. In comparison, G^{2}Miner significantly improves the warp execution efficiency. This is mainly due to the highly efficient implementation of our warp-centric set operations. Besides, we also measure *branch efficiency*, i.e., the ratio of non-divergent branches to total branches. Although G^{2}Miner uses DFS, We find that Pangolin and G^{2}Miner have almost the same branch efficiency, thanks to the two-level parallelism scheme. Since we assign each task to a warp, all threads in a warp does the same DFS walk synchronously, which avoids most of the branch divergence. This creates some redundancy, but since most of execution time is spent on set operations, it is still a good tradeoff.

Counting-only pruning. In §8.1, we do not enable optimization D in Table 2, because GraphZero and Pangolin do not support it. We observe that for those patterns (e.g., `diamond`) enabling this pruning in G^{2}Miner further improve performance by 6.2× on average. Enabling this optimization in Peregrine also improves its performance, as shown in Table 9. However, due to our high efficiency on GPU, G^{2}Miner still outperforms Peregrine by 41.1× when both enable it. This again demonstrates the performance superiority of GPU over CPU, no matter what algorithm optimizations are applied.

Sorting and renaming vertices. For fair comparison, this optimization done by the preprocessor is also not enabled in §8.1. Our evaluation shows that this can futher improve G^{2}Miner performance by 5% (up to 90%). Applying this to GraphZero also helps, but G^{2}Miner is still 12× faster.

9 Conclusion

We present G^{2}Miner, the first multi-GPU GPM framework that supports efficiently mining large graphs and patterns. For high efficiency, G^{2}Miner is aware of the input, pattern and architecture to fully unlock the potential of GPM computing on GPUs, which results in a 5× speedup over the state-of-the-art GPU-based GPM system, Pangolin, on a single GPU. For scalability, G^{2}Miner employs a custom task scheduler that can scale GPM computation to multiple GPUs linearly. For programmability, it automatically enables applicable optimizations and generates CUDA code, which hides away GPU programming complexity, and in turn provides the same easy-to-use programming interface as the state-of-the-art CPU-based GPM frameworks (e.g., Peregrine). We also show that G^{2}Miner on a single V100 GPU is 48× faster than Peregrine on a 56-core Intel CPU, a free lunch for GPM users.

10 Acknowledgements

This research is funded by Samsung Semiconductor (GRO grants) and MIT-IBM Watson AI Lab, and supported by XSEDE allocation TG-CIE-170005 and ASC22045. We thank Tianhao Huang and OSDI reviewers for their feedback.

References

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 61:1–61:12, Piscataway, NJ, USA, 2016. IEEE Press.

[2] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4), October 2017.

[3] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *ICDM*, pages 1–10, 2015.

[4] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery*, 29(3):626–688, 2015.

[5] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. K-clique counting on gpus. *arXiv preprint arXiv:2104.13209*, 2021.

[6] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and SC. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):241–249, 2008.

[7] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704, February 2018.

[8] Rasmus Resen Amossen and Rasmus Pagh. A new data layout for set intersection on gpus. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 698–708. IEEE, 2011.

[9] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24, 2008.

[10] Christos Bellas and Anastasios Gounaris. An evaluation of large set intersection techniques on gpus. In *DOLAP*, pages 111–115, 2021.

[11] Austin R. Benson, David F. Gleich, and Jure Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[12] Indrajit Bhattacharya and Lise Getoor. Entity resolution in graphs. *Mining graph data*, 311, 2006.

[13] Bibek Bhattarai, Hang Liu, and H. Howie Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1447–1462, New York, NY, USA, 2019. ACM.

[14] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1199–1214, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Jovan Blanuša, Radu Stoica, Paolo Ienne, and Kubilay Atasu. Manycore clique enumeration with fast set intersections. *Proc. VLDB Endow.*, 13(12):2676–2690, July 2020.

[16] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[17] Vincenzo Bonnici, Rosalba Giugno, and Nicola Bombieri. An efficient implementation of a subgraph isomorphism algorithm for gpus. In *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 2674–2681. IEEE, 2018.

[18] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, Nov 2012.

[19] Amlan Chatterjee, Sridhar Radhakrishnan, and John K. Antonio. Counting problems on graphs: Gpu storage and parallel computing techniques. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 804–812, 2012.

[20] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: An efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Jingji Chen and Xuehai Qian. Dwarvesgraph: A high-performance graph mining system with pattern decomposition, 2021.

[22] Jingji Chen and Xuehai Qian. Kudu: An efficient and scalable distributed graph pattern mining engine, 2021.

[23] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. Dynamic load balancing on single-and multi-gpu systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[24] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive cache management for energy-efficient gpu computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 343–355, Washington, DC, USA, 2014. IEEE Computer Society.

[25] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining. In *Proceedings of the 35th ACM International Conference on Supercomputing*, ICS '21, 2021.

[26] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(8), August 2020.

[27] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind. Flexminer: A pattern-aware accelerator for graph pattern mining. In *Proceedings of the International Symposium on Computer Architecture*, 2021.

[28] X. Cheng, C. Dale, and J. Liu. Dataset for statistics and social network of youtube videos. http://netsg.cs.sfu.ca/youtubedata/.

[29] Young-Rae Cho and Aidong Zhang. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *IEEE Transactions on information technology in biomedicine*, 14(1):30–36, 2009.

[30] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs*. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 589–598, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[31] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1036–1050, Aug 2005.

[32] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 393–404, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1357–1374, New York, NY, USA, 2019. ACM.

[34] Carmel Domshlak, Samir Genaim, and Ronen Brafman. Preference-based configuration of web page content. In *14th European Conference on Artificial Intelligence (ECAI 2000), Configuration Workshop, Berlin, Germany*, pages 19–22, 2000.

[35] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, March 2014.

[36] Katherine Faust. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks*, 32(3):221 – 233, 2010.

[37] Dima Feldman and Yuval Shavitt. Automatic large scale generation of internet pop level maps. In *IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1–6. IEEE, 2008.

[38] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A Bader. Fast and adaptive list intersections on the gpu. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[39] I. Giechaskiel, G. Panagopoulos, and E. Yoneki. PDTL: Parallel and distributed triangle listing for massive graphs. In *2015 44th International Conference on Parallel Processing*, pages 370–379, Sep. 2015.

[40] Oded Green, James Fox, Alex Watkins, Alok Tripathy, Kasimir Gabert, Euna Kim, Xiaojing An, Kumar Aatish, and David A Bader. Logarithmic radix binning and vectorized triangle counting. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[41] Oded Green, Robert McColl, and David A. Bader. Gpu merge path: A gpu merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 331–340, New York, NY, USA, 2012. ACM.

[42] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page

1067–1082, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Wentian Guo, Yuchen Li, and Kian-Lee Tan. Exploiting reuse for gpu subgraph enumeration. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.

[44] B. H. Hall, Jaffe A. B., and Trajtenberg M. The NBER patent citation data file: Lessons, insights and methodological tools. http://www.nber.org/patents/, 2001.

[45] Shuo Han, Lei Zou, and Jeffrey Xu Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1587–1602, 2018.

[46] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turbo_{iso}: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 337–348, New York, NY, USA, 2013. Association for Computing Machinery.

[47] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. DistTC: High performance distributed triangle counting. In *HPEC 2019 23rd IEEE High Performance Extreme Computing, Graph Challenge*, September 2019.

[48] Paul W Holland and Samuel Leinhardt. Local structure in social networks. *Sociological methodology*, 7:1–45, 1976.

[49] Lin Hu, Lei Zou, and Yu Liu. Accelerating triangle counting on gpu. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD-/PODS '21, page 736–748, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Y. Hu, H. Liu, and H. H. Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182, Nov 2018.

[51] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proc. VLDB Endow.*, 8(3):293–304, November 2014.

[52] Shweta Jain and C. Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 441–449, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[53] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth EuroSys Conference*, EuroSys '20, 2020.

[54] Madhav Jha, C. Seshadhri, and Ali Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 495–505, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.

[55] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1695–1698, New York, NY, USA, 2017. Association for Computing Machinery.

[56] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[57] Hisashi Kashima, Hiroto Saigo, Masahiro Hattori, and Koji Tsuda. Graph kernels for chemoinformatics. In *Chemoinformatics and advanced machine learning perspectives: complex computational methods and collaborative techniques*, pages 1–15. IGI Global, 2011.

[58] Robest Kessl, Nilothpal Talukder, Pranay Anchuri, and Mohammed J. Zaki. Parallel graph mining with gpus. In *Proceedings of the 3rd International Conference on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications - Volume 36*, BIGMINE'14, pages 1–16. JMLR.org, 2014.

[59] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. DUALSIM: Parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1231–1245, New York, NY, USA, 2016. ACM.

[60] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 411–426, New York, NY, USA, 2018. ACM.

[61] Daphne Koller, Nir Friedman, Sašo Džeroski, Charles Sutton, Andrew McCallum, Avi Pfeffer, Pieter Abbeel, Ming-Fai Wong, Chris Meek, Jennifer Neville, et al. *Introduction to statistical relational learning*. MIT press, 2007.

[62] Oleksii Kuchaiev, Tijana Milenković, Vesna Memišević, Wayne Hayes, and Nataša Pržulj. Topological network alignment uncovers biological function and phylogeny. *Journal of the Royal Society Interface*, 7(50):1341–1354, 2010.

[63] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[64] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[65] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *Proc. VLDB Endow.*, 8(10):974–985, June 2015.

[66] J. Leskovec. Snap: Stanford network analysis platform, 2013.

[67] W. Lin, X. Xiao, X. Xie, and X. Li. Network motif discovery: A gpu approach. In *2015 IEEE 31st International Conference on Data Engineering*, pages 831–842, April 2015.

[68] Xiaojie Lin, Rui Zhang, Zeyi Wen, Hongzhi Wang, and Jianzhong Qi. Efficient subgraph matching using gpus. In Hua Wang and Mohamed A. Sharaf, editors, *Databases Theory and Applications*, pages 74–85, Cham, 2014. Springer International Publishing.

[69] Yuan Lin and Vinod Grover. Using cuda warp-level primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/, 2018.

[70] Yongchao Liu, Bertil Schmidt, Weiguo Liu, and Douglas L. Maskell. Cuda-meme: Accelerating motif discovery in biological sequences using cuda-enabled graphics processing units. *Pattern Recognition Letters*, 31(14):2170 – 2177, 2010.

[71] Bruce T Lowerre. *The harpy speech recognition system.* Carnegie Mellon University, 1976.

[72] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. Distributed graph pattern matching. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 949–958, New York, NY, USA, 2012. ACM.

[73] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. Graphzero: A high-performance subgraph matching system. *SIGOPS Oper. Syst. Rev.*, 55(1):21–37, June 2021.

[74] Daniel Mawhirter and Bo Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 509–523, New York, NY, USA, 2019. ACM.

[75] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, July 2019.

[76] Tijana Milenković, Weng Leong Ng, Wayne Hayes, and Nataša Pržulj. Optimal network alignment with graphlet degree vectors. *Cancer informatics*, 9:CIN–S4744, 2010.

[77] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[78] Caleb C Noble and Diane J Cook. Graph-based anomaly detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2003.

[79] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge path-parallel merging made simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1611–1618. IEEE, 2012.

[80] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. H-index: Hash-indexing for parallel triangle counting on gpus. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, Sep. 2019.

[81] Roger Pearce, Trevor Steil, Benjamin W Priest, and Geoffrey Sanders. One quadrillion triangles queried on one million processors. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2019.

[82] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 1431–1440, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[83] Natasa Pržulj, Derek G Corneil, and Igor Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[84] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. Graph kernels for chemical informatics. *Neural networks*, 18(8):1093–1110, 2005.

[85] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6, 2014.

[86] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. Fast and robust distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 12(11):1344–1356, 2019.

[87] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-aware warp scheduling. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 99–110, 2013.

[88] Ryan A Rossi and Nesreen K Ahmed. Role discovery in networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):1112–1131, 2014.

[89] Ryan A. Rossi and Rong Zhou. Leveraging multiple gpus and cpus for graphlet counting in large networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 1783–1792, New York, NY, USA, 2016. ACM.

[90] Satu Elisa Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[91] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 625–636, New York, NY, USA, 2014. ACM.

[92] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. *arXiv preprint arXiv:2002.10047*, 2020.

[93] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[94] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160, April 2015.

[95] Shuya Suganami, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Accelerating all 5-vertex subgraphs counting using gpus. In *International Conference on Database and Expert Systems Applications*, pages 55–70. Springer, 2020.

[96] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. Efficient parallel subgraph enumeration on a single machine. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 232–243. IEEE, 2019.

[97] Shixuan Sun and Qiong Luo. Scaling up subgraph query processing with efficient subgraph matching. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 220–231. IEEE, 2019.

[98] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM.

[99] N. Talukder and M. J. Zaki. A distributed approach for graph mining in massive networks. *Data Min. Knowl. Discov.*, 30(5):1024–1052, September 2016.

[100] N. Talukder and M. J. Zaki. Parallel graph mining with dynamic load balancing. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3352–3359, Dec 2016.

[101] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, New York, NY, USA, 2015. ACM.

[102] Tatsuya Toki and Tomonobu Ozaki. Experimental evaluation of a gpu-based frequent subgraph miner using synthetic databases. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, pages 504–507, 2016.

[103] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. Fast subgraph matching on large graphs using graphics processors. In Matthias Renz, Cyrus Shahabi, Xiaofang Zhou, and Muhammad Aamir Cheema, editors, *Database Systems for Advanced Applications*, pages 299–315, Cham, 2015. Springer International Publishing.

[104] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An almost Depth-First-Search distributed Graph-Querying system. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 209–224. USENIX Association, July 2021.

[105] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[106] Fei Wang, Jianqiang Dong, and Bo Yuan. Graph-based substructure pattern mining using cuda dynamic parallelism. In Hujun Yin, Ke Tang, Yang Gao, Frank Klawonn, Minho Lee, Thomas Weise, Bin Li, and Xin Yao, editors, *Intelligent Data Engineering and Automated Learning – IDEAL 2013*, pages 342–349, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[107] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 763–782, Berkeley, CA, USA, 2018. USENIX Association.

[108] Leyuan Wang and John D Owens. Fast gunrock subgraph matching (gsm) on gpus. *arXiv preprint arXiv:2003.01527*, 2020.

[109] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8, 2016.

[110] Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. Are dynamic memory managers on gpus slow? a survey and benchmarks. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 219–233, New York, NY, USA, 2021. Association for Computing Machinery.

[111] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[112] Di Wu, Fan Zhang, Naiyong Ao, Fang Wang, Xiaoguang Liu, and Gang Wang. A batched gpu algorithm for set intersection. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 752–756. IEEE, 2009.

[113] Di Wu, Fan Zhang, Naiyong Ao, Gang Wang, Xiaoguang Liu, and Jing Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[114] Xifeng Yan and Jiawei Han. gspan: graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, pages 721–724, Dec 2002.

[115] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[116] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V Çatalyürek. Fast triangle counting using cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[117] Li Zeng, Lei Zou, M Tamer Özsu, Lin Hu, and Fan Zhang. Gsi: Gpu-friendly subgraph isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1249–1260. IEEE, 2020.

[118] Jiyuan Zhang, Yi Lu, Daniele G Spampinato, and Franz Franchetti. Fesia: A fast and simd-efficient set intersection approach on modern cpus. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1465–1476. IEEE, 2020.

[119] Luming Zhang, Mingli Song, Zicheng Liu, Xiao Liu, Jiajun Bu, and Chun Chen. Probabilistic graphlet cut: Exploiting spatial structure cue for weakly supervised image segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1908–1915, 2013.

[120] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. In *Proceedings of the 2020 IEEE International Conference on Data Engineering (ICDE 2020)*, ICDE '20, 2020.

A Artifact Appendix

Abstract

This artifact appendix helps the readers reproduce the main evaluation results of the OSDI' 22 paper: Efficient and Scalable Graph Pattern Mining on GPUs.

Scope

The artifact can be used for evaluating and reproducing the main results of the paper, including Table 4, Table 5, Table 6, Table 7, Table 8 and Fig. 9, Fig. 10, Fig. 11, Fig. 12 in §8.

Contents

The artifact evaluation includes all the experiments in the paper. Details of the experiments are listed here: https://github.com/chenxuhao/GraphMiner/blob/master/OSDI-experiments-guide.md

Hosting

The source code of this artifact can be found on GitHub: https://github.com/chenxuhao/GraphMiner, master branch.

Requirements

Hardware dependencies

This artifact depends on an NVIDIA V100 GPU.

Software dependencies

This artifact requires CUDA toolkit 11.1.1 or greater and GCC 8 or greater.

Details of the dependencies are listed here: https://github.com/chenxuhao/GraphMiner/blob/master/OSDI-experiments-guide.md